6800 SOFTWARE GOURMET GUIDE & COOKBOOK

ROBERT FINDLEY

PUT TOGETHER YOUR OWN 6800 PROGRAMS USING THESE TIME-TESTED RECIPES: UTILITY ROUTINES • CONVERSION ROUTINES • INPUT/OUTPUT ROUTINES • SEARCH AND SORT ROUTINES • FLOATING POINT ROUTINES • 6800 INSTRUCTION SET • AND MORE

HAYDEN

6800 SOFTWARE GOURMET GUIDE & COOKBOOK

ROBERT FINDLEY

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

ACKNOWLEDGEMENT

The author wishes to thank the following members of the staff at Scelbi Computer Consulting, Inc., for their assistance in the preparation of this book.

Nat Wadsworth
Raymond Edwards

Printed in the United States of America

1 2 3 4 5 6 7 8 9 PRINTING

82 83 84 85 86 87 88 89 90 YEAR

****
CONTENTS
****

INTRODUCTION

Have you tried cooking up a program lately on your 6800 microcomputer, and you just can't seem to get the right mixture of instructions? Or did that math recipe your friend gave you turn out to have too many bugs in it, and leave a sour taste in your mouth? Don't toss your computer in the sink and grind those bad listings up in the garbage disposal. Here's a book that will help take you from a novice that burns the bits to a gourmet chef that can make the sweetest APPLEcations program pie imaginable.

Before throwing together your favorite dish, a thorough knowledge of the basic ingredients, namely the 6800 instruction set, is essential. Every chef that's worth his salt knows exactly what each ingredient will do for him. Begin creating your masterpiece by mixing in a little of this routine and a little of that routine. Spice up the program with a few of your own special application routines, and before baking, add a personal touch by folding in the input/output driver routines for the peripherals in your system. Bake thoroughly with your assembler, and there you have it! Your programming masterpiece, ready to feed into your computer's memory for hours of tasty enjoyment.

Is your taste for math routines? Or manipulating data tables and character strings? Or maybe you wish to do some real time programming. Or set up your system to operate the peripherals under interrupt control. Whatever your requirements may be, there is certain to be some ideas, techniques, and routines in this book to aid you in programming for your specific application.

THE 6800 INSTRUCTION SET

The instruction set of the 6800 CPU provides considerable programming power to the machine language programmer. There are 72 basic instructions which, when all permutations are considered, provide 197 individual instructions. These instructions use from one to three bytes of memory depending on the function they perform.

There are several basic elements in the structure of the 6800 CPU that the programmer must become thoroughly familiar with. These elements include the Program Counter, two Accumulators, the Index Register, the Stack Pointer, Memory, and the Status Flags. Also, concepts such as performing inputs and outputs using the same instructions which access memory, the several modes of interrupt operations possible, and the various addressing modes utilized by this instruction set should be thoroughly understood before attempting to write machine language programs.

The Program Counter is a sixteen-bit register which is used to direct the flow of a program from one instruction to another. Since the program counter is sixteen bits long, it can directly access instructions in any of the possible 64 K bytes of memory. After an instruction is executed, the program counter is automatically incremented to the next location in memory from which the next instruction to be executed will be taken, unless the current instruction directs the computer to a different memory location, such as a jump, branch, or return instruction.

The two accumulators of the 6800 are the real workhorse elements from the software point of view. These accumulators are eight-bit registers and are given the arbitrary designations of the A accumulator and the B accumulator. The capability of these accumulators is essentially the same. All arithmetic and boolean logic operations "accumulate" their results in one of these registers. Also, most data transfers from one memory location to another use one of these accumulators as intermediate storage. There are also a number of shift, rotate, and comparing operations that can be done with the contents of these accumulators. In addition, the condition of the status flags is affected by almost every operation with the accumulators. The A accumulator has several instructions associated with it which give it slightly greater capability than the B accumulator. Addition, subtraction, and comparing operations between these two

accumulators may only be accomplished by using A as the accumulator.

The Index register is a sixteen-bit register that is used primarily as a memory pointer. It may be used to point to memory locations on which various memory to accumulator instructions operate. Or, it may indicate the table or storage area for transferring data from memory to accumulator or accumulator to memory. It may also be used as a count down or count up register, since the condition of the zero status flag will indicate when the index register has been incremented or decremented to zero. Instructions that use the index register as a memory pointer also allow a displacement to be defined which is added to the address in the index register in determining which memory location is to be used by the instruction.

The Stack Pointer is a sixteen-bit register that must be set up to point to an area in memory to be used as the STACK. The STACK is the storage area in which the 6800 CPU saves the return addresses of subroutine calls and the pertinent data that must be stored when an interrupt occurs. Therefore, the stack pointer must designate an area in RAM memory for use as the stack. The data is stored and retrieved from the stack in a push-pull manner. This method of stack operation will be discussed in greater detail later. The stack may also be used to temporarily store the contents of the A and B accumulators.

The Memory is the element in which the programs to be executed are stored, along with the storage of data that may be used by the programs. The 6800 is capable of directly addressing up to 64 K of memory. Each memory location consists of eight bits, which together are referred to as a byte. The memory associated with any one individual system may vary. It may consist of a combination of ROM and/or PROM memory, which contains permanently stored programs or data, or RAM memory, whose contents may be altered by the computer for storing various programs or data as needed.

The input/output structure of the 6800 allows the transfer of data to and from the peripheral interfaces by assigning memory addresses to the peripheral. By setting up memory locations as the channels through which the data is transferred to and from the peripherals, it is possible to use any of the instructions that refer to memory for transferring the I/O data. This affords the programmer great flexibility in testing the status and controlling the peripheral devices.

In order to make decisions based on the contents of a register or memory location, or the results of an arithmetic or logical operation, the 6800 has six status, or condition, flags. These status flags are set to one (for a true condition), or cleared to zero (for a false condition) in accordance with the results of an operation performed. Not all status flags are affected by the execution of each instruction. Only those flags that have relevance to the result of the operation are affected by an instruction. These status flags are referred to as the Carry (C), Half carry (H), Interrupt (I), Negative (N), Overflow (V), and Zero (Z) flags. The condition of all but the Half carry and the Interrupt flags may be tested by several instructions, and the instructions' operation may vary as a consequence of the flags particular status at the time it is tested.

The Carry flag may be considered an extension of the eight bit accumulator or a memory location used as the operand of an instruction. For addition and subtraction operations, the carry is considered the ninth bit and will indicate when an addition causes an overflow from bit seven, or a subtraction requires a borrow for bit seven. By functioning in this manner, the carry flag becomes a necessary link when performing multiple-precision operations. The carry flag is also considered an extension either to the left or the right of a register or memory location in various rotate and shift operations. There are a number of instructions that set up the carry to a given condition. This function may be necessary when performing a group of instructions that require the carry to be initially set to a known state.

The Half carry flag is used to indicate when an overflow or underflow from bit three occurs as a result of an addition operation. Unlike the other status flags, this cannot be tested by a program. The only function of relevance for this flag is in the operation of the decimal adjust accumulator instruction (DAA), which adjusts the accumulator from its binary contents to two "BCD" digits, in a manner to be defined later in this chapter. The presence of the half carry flag and the DAA instruction provides the 6800 with the capability to perform decimal arithmetic without conversion to binary.

The Interrupt flag is used to indicate when the maskable interrupt is disabled. When this flag is set to one, the maskable interrupt input is disabled, and the CPU will not respond to an interrupt on this line. When this flag is cleared, an interrupt on the maskable interrupt line will be acknowledged by the CPU. This flag is set upon receipt of any

one of the three interrupts; upon returning from the interrupt, it is restored to its initial condition at the time the interrupt was received. It may also be set or cleared by the execution of two instructions that perform this specific function.

The Negative flag indicates the condition of the most significant bit of the register or memory location after the execution of the last instruction that affects the negative flag. If the result leaves the most significant bit set to one, the negative flag will be set to one. If the most significant bit is zero, the negative flag will also be zero. For example, if the addition of the contents of accumulator B to the contents of accumulator A results in the most significant bit in accumulator A being set to one, the negative flag will be set to one. Or, if a memory location is rotated once to the right, moving a zero into the most significant bit, the negative flag will be cleared to zero as a result of the operation.

The Overflow flag provides an indication of a two's complement overflow as a result of an addition or subtraction. For addition, the two's complement overflow occurs when bit seven of both addends is the same value and bit seven of the sum is the opposite value, i.e., the addition of two negative numbers equalling a positive value. For subtraction, a two's complement overflow occurs when bit seven of the subtrahend and minuend are opposite, and bit seven of the result takes on the value of bit seven of the subtrahend, i.e., the subtraction of a negative number from a positive number with a negative result.

The Zero flag is set to one when the execution of an instruction results in an all zero value. This may occur following an arithmetic or Boolean logic operation. It may also occur after an accumulator, or the index register or the stack pointer register, or a memory location has been incremented or decremented to zero.

These status flags are arranged in an eight bit register. The two most significant bits of this register are always set to one, and the remaining six bits contain the status flags as detailed below. The flags are arranged in this single register so that they may be easily stored and retrieved for interrupt operations. This register is often referred to as the CONDITION CODE register.

The stack is used to store and retrieve data in the memory loca-

CONDITION CODE REGISTER
BIT DEFINITION

BIT 0	-	CARRY FLAG	C
BIT 1	-	OVERFLOW FLAG	V
BIT 2	-	ZERO FLAG	Z
BIT 3	-	NEGATIVE FLAG	N
BIT 4	-	INTERRUPT FLAG	I
BIT 5	-	HALF CARRY FLAG	H

tions indicated by the stack pointer. The stack pointer operates in a push-pull manner. Its operation is the same whether the data being stored is a return address from a subroutine call, or the register contents at the time of an interrupt, or the storage or retrieval of the contents of either accumulator. When data is stored in the stack, the data byte is stored in the memory location indicated by the stack pointer, and the stack pointer is then automatically decremented. If more than one byte is to be stored, as in the storage of a return address or at the time of an interrupt, each additional byte is loaded into memory and the stack pointer is decremented following each byte storage. By automatically decrementing the stack pointer in this manner, it is positioned to store more data or read data stored in the stack when either a pull instruction or a return from subroutine or interrupt is executed. The following illustrates the method of storing the return address of a subroutine call in the stack. The return address to be stored is location 5E on page 02.

BEFORE SUBROUTINE CALL:

STACK POINTER	MEMORY ADDRESS OF STACK	STACK CONTENTS
00FF	00FD	00
	00FE	00
	00FF	00

AFTER SUBROUTINE CALL:

00FD	00FD	00
	00FE	02
	00FF	5E

When data is read from the stack, by performing a return or pull instruction, the reverse procedure is followed. That is, the stack pointer is automatically incremented and the data byte is then read from the stack. The stack pointer is now positioned for the next stack operation, whether it be to read or write data in the stack.

The 6800 CPU has provisions for three types of interrupts. Two interrupts are generated by hardware, and the third is an interrupt created by a software instruction. The CPU responds to each of these interrupts by storing the contents of all the CPU registers, namely, the condition code register, the A and B accumulators, the index register, and the program counter in the stack. (This data storage procedure may be performed previous to a hardware interrupt by the execution of a "Wait for interrupt" instruction.) The CPU then selects an interrupt vector according to the type of interrupt received. This interrupt vector is actually a start address for an interrupt service routine. In most cases, this interrupt service routine begins in ROM memory with several short instructions that fetch another address set up in RAM memory by the programmer. This second address would be the start of the actual interrupt service routine written to operate the devices associated with one's system.

The two interrupts initiated by hardware are referred to as non-maskable interrupts and maskable interrupts. The non-maskable interrupt, when received, will always be acknowledged by the CPU. This type of interrupt would be required for high speed devices that have a very short amount of time to transfer the data. On the other hand, acknowledgement of a maskable interrupt is controlled by the setting of the interrupt status flag. As previously discussed, when this flag is reset to zero, the CPU will acknowledge a maskable interrupt. If this flag is set to a one, an interrupt on the maskable interrupt line will not be acknowledged by the CPU. This allows the programmer to control when the program can and cannot accept an interrupt on the maskable interrupt line. Each of these hardware interrupts has a separate vector assigned to it.

A software interrupt is generated by the execution of the software interrupt instruction. The 6800 reacts to the CPU in the same fashion as it would to a non-maskable interrupt. There is also a separate interrupt vector used for a software interrupt. This eliminates the need for the interrupt service routine to determine whether the interrupt was a hardware generated or software generated interrupt.

The use of interrupts in a microcomputer system allows a program to be performing one function while waiting for a peripheral device to complete its operation. For example, a mailing list program could be sorting out names of people living in a specific geographical area, while a printer device, operating under interrupt control, prints the selected names.

There is also a RESET interrupt which is generally used to direct the computer to a built-in monitor routine, such as the MIKBUG** program. This reset interrupt does not perform the CPU register storage as performed by the other interrupts. It is simply an overriding interrupt that stops whatever program execution may be in progress and directs control to the resident monitor routine. A separate vector is assigned for the RESET interrupt just as the vectors for the other interrupts.

The interrupt vectors are set up in the hardware at the highest addressable locations of the computer (generally FFF8-FFFF). As discussed, these vectors direct the CPU to specific memory locations when the respective interrupts occur. The page portion of the vector address is in the lower address, and the low portion of the vector is in the higher address of each vector. The vectors are arranged in memory as follows:

ADDRESS OF VECTOR	TYPE OF INTERRUPT
FFFF,FFFE	RESET
FFFD,FFFC	Non-Maskable
FFFB,FFFA	Software Interrupt
FFF9,FFF8	Maskable

The 6800 instruction set makes extensive use of various ADDRESSING modes. These different modes of addressing provide many of the instructions with two, three, or four ways of selecting the operand with which the instruction is to execute. The addressing mode may refer to the location that contains (or is to receive) data for the instruction execution. Or, it may refer to the location of the next instruction to be executed. The instructions that use these different addressing modes require an additional one or two bytes of

**MIKBUG is a registered trademark of Motorola Corporation

memory to be properly defined by the actual machine code. The first byte of the instruction contains the machine code which indicates the instruction to be executed along with the addressing mode used for that instruction. The information contained in the additional bytes of the instruction would indicate either the actual data to be used as the operand, or the location in memory where the data is (or will be) stored, or a relative address with respect to either the program counter or index register. These addressing modes, to be described next, are referred to as immediate addressing, direct addressing, extended addressing, relative addressing, and indexed addressing.

The source listing of the instructions that use these modes is separated into two fields. The first field is called the operator field, and contains the mnemonic for the operation to be performed. The second field is the operand field which will indicate the addressing mode to be used for the instruction. As will be pointed out later in this chapter when the individual instructions are presented, the machine code for the same mnemonic will vary, depending on the addressing mode selected.

Whenever a numeric value is designated as the operand of the source listing for an instruction, the value will be represented by hexadecimal digits. In order to conform with the generally accepted notation for representing hexadecimal values in the source listing, these values will be preceeded by a dollar sign ($). For example, an instruction to load accumulator B from memory location 00A7 will appear as follows:

```
LDAB $00A7
```

The immediate addressing mode selects the operand from the memory location following the first byte of the instruction. The instructions that allow the immediate mode of addressing require two bytes when the operator is either the A or B accumulator. For example, the "load accumulator A" instruction in the immediate mode takes the contents of the byte following the machine code for the instruction and loads it into the A accumulator. There are three instructions that use three bytes of memory for the immediate mode of addressing. These three instructions allow one to load the index register or stack pointer with an immediate value, or compare the contents of the index register against the immediate value.

The additional two bytes are required by these instructions because the registers they operate on are sixteen bits in length. In the listings contained in this book, whenever the immediate mode of addressing is used, the operand will be preceeded by a pound sign (#).

The following example illustrates the execution of the instruction that loads the A accumulator with the immediate value of ten.

BEFORE EXECUTION:
Contents of A = XX (don't care)

INSTRUCTION EXECUTED
Source Code LDAA #$10 Machine Code 86 10

AFTER EXECUTION:
Contents of A = 10

The direct addressing mode selects the operand of the instruction from a memory location on page 00. This mode requires one additional byte to specify the location on page 00 to be used by the instruction. This mode of addressing makes it advantageous to use page 00 for the storage of frequently used data, as it allows one to access the specific location on page 00 with a two-byte instruction rather than having to use an additional byte to specify the page, as in the extended mode.

The example below illustrates the execution of the store the A accumulator instruction using the direct addressing mode. The instruction in the example stores the contents of the A accumulator in memory location 49.

BEFORE EXECUTION:
Contents of A = 85
Contents of memory location 0049 = XX (don't care)

INSTRUCTION EXECUTED
Source Code STAA $49 Machine Code 97 49

AFTER EXECUTION:
Contents of A = 85
Contents of memory location 0049 = 85

The extended addressing mode uses two additional bytes to define the address of the memory location to be used as the operand for the instruction. The first of these two bytes contains the page portion of the memory address; the second contains the lower portion. Thus, the extended mode allows one to directly access any memory location in the system for use as the operand of the instruction. When instructions that allow both extended and direct addressing modes are assembled, the distinction between the two is determined by the page number of the address. If the page number is zero, the direct addressing mode should be selected. If the page number is not zero, the extended addressing mode must be used.

The following example illustrates the execution of the load the A accumulator with the contents of a memory location using the extended addressing mode. In the example, the contents of memory location 0180 are loaded into the A accumulator.

BEFORE EXECUTION:
Contents of A = XX (don't care)
Contents of memory location 0180 = 67

INSTRUCTION EXECUTED
Source Code LDAA $0180 Machine Code B6 01 80

AFTER EXECUTION:
Contents of A = 67
Contents of memory location 0180 = 67

The relative addressing mode references a memory location "relative" to the current value of the program counter +2. The relative addressing mode is used exclusively by the branch instructions. The current value of the program counter, used to calculate the displacement for the branch instructions, is the memory location in which the first byte of the branch instruction resides. One additional byte is required for the branch instructions. This byte contains the relative displacement in a two's complement form. The memory location to be branched to is calculated by simply adding the second byte to the value of the program counter +2. If the most significant bit is a one, the branch will be to an address lower than the current program counter +2. A value of zero for the most significant bit indicates a branch to a higher address. The two's complement nota-

tion limits the branch instructions to a displacement of -128 to +127 locations from the value of the program counter +2.

The following example illustrates a branch back to the instruction located 0E hexadecimal locations before the branch instruction.

BEFORE EXECUTION:
Program Counter = 0170
(Location of 1st machine code of branch)

INSTRUCTION EXECUTED
Source Code BRA $F0 Machine Code 20 F0

AFTER EXECUTION:
Program counter = 0162

The indexed addressing mode uses the current contents of the index register plus a forward displacement to define the memory location for the operand of the instruction. The forward displacement is stored in the second byte of an index instruction. This displacement is defined in unsigned binary notation. That is, the eight bit displacement value is added to the contents of the index register providing a range of 0 to +255 locations from the value in the index register. Using this mode, one can set the index register to a set location, such as the start of a data area, and reference various locations in that area by defining the proper displacement. In the source listings used throughout this book, an index instruction will be indicated by a single "X" or a "disp,X" entry in the operand field.

The example below illustrates the use of the indexed addressing mode. The instruction to load the A accumulator in the indexed mode is used. One should note the use of the forward displacement which is added to the index register to designate the memory location.

BEFORE EXECUTION:
Contents of A = XX (don't care)
Contents of index register = 0150
Contents of memory location 0150 = AA
Contents of memory location 0151 = BB
Contents of memory location 0152 = CC
Contents of memory location 0153 = DD

INSTRUCTION EXECUTED
Source Code LDAA $02,X Machine Code A6 02

AFTER EXECUTION:
Contents of A = CC
Contents of index register = 0150

The following description of the various types of instructions available with the 6800 CPU will provide the mnemonic name used for writing programs in symbolic language, along with the machine code for the instruction given as two hexadecimal digits. In cases where the mnemonic allows more than one addressing mode, the additional machine codes will be listed, followed by an indication of the addressing mode they relate to. Appendix A contains a list of these mnemonics and machine codes in alphabetical order. These mnemonics are equivalent to those defined by Motorola. Additionally, information concerning the timing for the instructions is included in this appendix. (A discussion on the significance of the timing for these instructions will be presented later in this chapter and at various points throughout the book.) If the programmer is not already aware of it, the use of mnemonics facilitates working with an assembler program when it is desired to develop relatively large, complex programs. Thus, the programmer is urged to concentrate on learning the mnemonics for the instructions, and not waste time memorizing the machine codes. After a program has been written using the mnemonics, the programmer can always use a look-up table to convert to machine code if an assembler program is not available. It's a lot easier technique (and subject to less error) than trying to memorize the 197 digital combinations that make up the machine code instruction set!

In the following discussion of the 6800 instruction set, each paragraph for a given instruction type is preceeded by the mnemonics and machine code in either two or three columns. The first column will contain the mnemonic representation of the instruction. The second column will contain the machine code for that mnemonic, and, in cases where several addressing modes are possible, the third column will indicate the addressing mode for the machine code.

The first group of instructions to be discussed loads data from one accumulator to the other, or from an accumulator to memory, and vice versa. These instructions require from one to three bytes of memory.

TRANSFER FROM ONE ACCUMULATOR TO THE OTHER

TAB	16
TBA	17

These two instructions transfer the contents of one accumulator to the other. The FROM accumulator is designated in the mnemonic by the second letter; the TO accumulator is designated by the third letter. Thus, the execution of the first instruction, TAB, transfers the contents of accumulator A to accumulator B. The second instruction, TBA, transfers the contents of accumulator B to lator A. The N and the Z flags will be conditioned to reflect the resulting contents of the TO accumulator, and the V flag will be cleared. The C, H, and I flags are not affected. These instructions require one byte of memory.

LOAD AN ACCUMULATOR FROM MEMORY

LDAA	#DATA	86	IMMEDIATE
LDAA	ADDR	96	DIRECT
LDAA	ADDR	B6	EXTENDED
LDAA	disp,x	A6	INDEXED
LDAB	#DATA	C6	IMMEDIATE
LDAB	ADDR	D6	DIRECT
LDAB	ADDR	F6	EXTENDED
LDAB	disp,x	E6	INDEXED

This group of instructions loads the designated accumulator, with the contents of the memory location indicated by the addressing mode. The accumulator is designated by the last letter in the mnemonic. For the immediate mode, the instruction requires two bytes and the data to be loaded into the accumulator is taken from the second byte of the instruction. For the direct mode, the instruction requires two bytes, with the second byte indicating the location on page 00 from which the data is to be taken and loaded into the accumulator. The second and third bytes of the three-byte extended mode instruction contain the page and low portion of the address from which the data to be loaded is taken. The index mode requires two bytes. The second byte contains the displacement which is added to the contents of the index register to calculate the memory address from which the data is taken. The N and Z flags are affected

as a result of the execution of these instructions. The V flag will be cleared, and the C, H, and I flags are not affected.

STORE AN ACCUMULATOR IN MEMORY

STAA	ADDR	97	DIRECT
STAA	ADDR	B7	EXTENDED
STAA	disp,X	A7	INDEXED
STAB	ADDR	D7	DIRECT
STAB	ADDR	F7	EXTENDED
STAB	disp,X	E7	INDEXED

Storing data contained in the designated accumulator to a memory location is accomplished by the execution of one of these instructions. The exact location in memory is determined by the addressing mode used. One should make note of the fact that the immediate mode is not valid for the two basic instructions. As in the load accumulator from memory instructions, the direct and indexed modes require two bytes, and the extended requires three. The status flags are also affected in a similar manner to the load the accumulator from memory instructions.

PUSH THE ACCUMULATOR ONTO THE STACK

PSHA	36
PSHB	37

These instructions store the contents of the designated register in the memory location pointed to by the stack pointer. After storing the data, the stack pointer is automatically decremented to properly position it for the next stack operation. These one-byte instructions provide a convenient method of temporarily storing the contents of either accumulator without designating specific memory locations for their storage. None of the status flags are affected by the execution of these instructions.

PULL DATA FROM THE STACK INTO THE ACCUMULATOR

PULA	32
PULB	33

The execution of these instructions first increments the stack pointer, and then transfers the data in the memory location indicated

by the stack pointer to the designated accumulator. These instructions are used in conjunction with the push instructions to retrieve data pushed onto the stack. The status flags are not affected by these pull instructions.

The next group of instructions deals with the manipulation of the contents of the accumulators. These one-byte instructions provide the capability to perform operations that would require either multiple byte instructions, or a sequence of several instructions to achieve the same result.

CLEAR THE ACCUMULATOR

CLRA	4F
CLRB	5F

These instructions clear the contents of the designated register by loading it with all zeros. The execution of these one-byte instructions clears the C, N, and V status flags, while setting the Z flag. The H and I flags are not affected.

INCREMENT THE ACCUMULATOR

INCA	4C
INCB	5C

These one-byte instructions increment the contents of the designated accumulator by one. Since the contents of the N, V, and Z flags are affected by the execution of these instructions, one may set up either accumulator as a counter, and branch to a routine out of the counting loop when the accumulator goes to zero, changes sign, or creates a two's complement overflow. The C, H and I flags are not affected by these instructions.

DECREMENT THE ACCUMULATOR

DECA	4A
DECB	5A

The contents of the designated accumulator are decremented by one by the execution of these instructions. The effect of these instructions on the status flags is the same as that for incrementing the accumulator instructions.

COMPLEMENT THE ACCUMULATOR

COMA	43
COMB	53

As one may already be aware, complementing a binary value is achieved by changing the digits that are initially one to a zero, and those that are zero to one. These one-byte instructions perform this operation on the contents of the designated accumulator. This same operation may be achieved by an EXCLUSIVE OR between the accumulator and a hexadecimal value of FF. The N and Z flags are affected by these instructions, and the V flag is cleared. The C, H, and I flags are not affected.

NEGATE THE ACCUMULATOR

NEGA	40
NEGB	50

These instructions change the contents of the designated accumulator to its two's complement form. The two's complement is achieved by subtracting the contents of the accumulator from zero. These one-byte instructions replace the three instruction sequence of storing the accumulator somewhere in memory, then clearing the accumulator, and finally, subtracting the initial stored value from zero. The two's complement may also be performed by complementing the value of the accumulator, and then incrementing by one. However, the condition of the C flag, as a result of the negate operation, reflects a subtraction from zero rather than the complement and increment operation. The C flag will be set except when the initial contents of the accumulator are zero. The N, V, and Z flags are also affected by these instructions. The H and I flags are not affected.

The next section contains instructions that deal with the loading, storing, and manipulation of the contents of the index register and stack pointer. Proper manipulation of these registers is essential in programming the 6800 efficiently. The number of bytes required for this group of instructions varies from one to three. Also, because these registers are sixteen bits long, the instructions which reference a memory address for loading or storing these registers actually refers to two bytes of memory. The first byte, at the address (M) called

out in the instruction, is used to load or store the page, or high, portion of the register. The next successive memory location (M+1) is used to load or store the low portion of the register.

LOAD THE INDEX REGISTER

LDX #ADDR	CE	IMMEDIATE
LDX ADDR	DE	DIRECT
LDX ADDR	FE	EXTENDED
LDX disp,X	EE	INDEXED

This group of instructions loads the index register from the memory location defined by the respective addressing mode. The immediate and extended instructions require three bytes of memory, while the direct and indexed require the usual two bytes. One should also note that the initial contents of the index register are used as a pointer in the indexed mode selecting its own new contents. The resultant contents of the index register affect the N and Z flags, and the V flag is cleared. The C, H, and I flags are not affected.

STORE THE INDEX REGISTER

STX ADDR	DF	DIRECT
STX ADDR	FF	EXTENDED
STX disp,X	EF	INDEXED

Storing the contents of the index register is accomplished by the execution of one of these instructions. The contents of the index register remain unchanged. One should note that there is no immediate mode of addressing for this instruction, since it would be of very little value. The direct and indexed mode require two bytes of memory for this instruction, and the extended requires three. The flags are affected in the same manner as for the load the index register instruction.

INCREMENT THE INDEX REGISTER

INX	08

This one byte instruction increments the index register by one. By setting up the index register as a pointer to a string of characters, this instruction may be used to advance the pointer, allowing one to fetch each character from the character string. Only the Z flag is

affected by the execution of this instruction. All other flags remain at their initial values.

DECREMENT THE INDEX REGISTER

DEX 09

This instruction decrements the index register by one. This one byte instruction simply performs the opposite function of the increment the index register instruction. Also, the Z flag is the only status flag affected by this instruction.

LOAD THE STACK POINTER

LDS	#ADDR	8E	IMMEDIATE
LDS	ADDR	9E	DIRECT
LDS	ADDR	BE	EXTENDED
LDS	disp,X	AE	INDEXED

This group of instructions loads the stack pointer register with the two byte memory contents indicated by the addressing mode. This basic instruction allows one to designate the area in memory to be used as the stack. The N and Z flags are affected by the results of this operation, and the V flag is cleared. The C, H, and I flags are not affected.

STORE THE STACK POINTER

STS	ADDR	9F	DIRECT
STS	ADDR	BF	EXTENDED
STS	disp,X	AF	INDEXED

The stack pointer contents are stored in the memory location (M) and the succeeding memory location (M+1) of the address designated by the addressing mode in these instructions. The stack pointer contents are not altered by the execution of these instructions. Making use of this basic instruction in conjunction with the load the stack pointer instruction, one may designate more than one area to be used in the manner of a stack. In some applications, this may allow more efficient use of memory since data may be transferred to and from the stack area by the single byte Push and Pull instructions. There is no immediate addressing mode provided for this instruction. The extended mode requires three bytes of memory, and the direct

and indexed require two. The N and Z flags are affected by the results of these instructions, and the V flag is cleared. The C, H, and I flags are not affected.

INCREMENT THE STACK POINTER

INS 31

This one-byte instruction increments the stack pointer by one. This instruction does not affect any of the status flags. This increment instruction, and the decrement instruction to be presented next, allows one to move the stack pointer up and down in the current stack area without altering the contents of the stack or the A and B accumulators.

DECREMENT THE STACK POINTER

DES 34

The stack pointer is decremented by one by the execution of this instruction. This instruction requires one byte and does not affect any of the status flags.

TRANSFER STACK POINTER TO INDEX REGISTER

TSX 30

This one-byte instruction transfers the contents of the stack pointer plus one to the index register. The stack pointer maintains its initial contents following the execution of this instruction. By loading the index register with the address contained in the stack pointer plus one, the index register is automatically set to the address of the last byte of data stored in the stack. This allows one to access the last byte of data stored in the stack using the indexed mode of addressing with a displacement of zero. This instruction does not affect any of the status flags.

TRANSFER THE INDEX REGISTER TO THE STACK POINTER

TXS 35

The contents of the index register minus one are loaded into the

stack pointer. The initial contents of the index register remain unchanged following the execution of this instruction. By decrementing the contents of the index register before loading it into the stack pointer, the stack pointer is automatically positioned to PULL the current data indicated by the index register from the stack. Also, it positions the stack pointer to allow data to be PUSHed onto the stack without destroying the data indicated by the index register. This instruction does not affect any of the status flags.

The instructions presented up to this point have discussed the transfer of data between internal CPU registers and a CPU register and a memory location. Several instructions that allow the manipulation of data within the CPU registers have also been discussed. The 6800 provides a similar type of manipulation of the contents of memory. These instructions utilize the extended and indexed mode of addressing, and, therefore, require two or three bytes of memory to properly define the instruction. This group of instructions is presented next.

CLEAR THE MEMORY CONTENTS

CLR ADDR	7F	EXTENDED
CLR disp,X	6F	INDEXED

This instruction clears the designated memory location by loading it with an all-zero byte. The extended addressing mode requires three bytes for the machine code of this instruction, and the indexed instruction requires two. This one byte instruction eliminates the necessity of loading an accumulator with zero and storing it in memory when one desires to clear a memory location. By using the index register as a memory pointer, a short program loop may be formed to clear out large areas of memory. This instruction clears the C, N, and V flags, and sets the Z status flag. The H and I flags are not affected.

INCREMENT THE MEMORY LOCATION

INC ADDR	7C	EXTENDED
INC disp,X	6C	INDEXED

The designated memory location is incremented by one by the execution of this instruction. The extended addressing mode requires three bytes of memory, and the indexed requires two bytes. This in-

struction makes it convenient to set up a memory location as a counter. The N, V, and Z status flags are conditioned to indicate the results of this operation on the memory contents. The C, H, and I flags are not affected.

DECREMENT THE MEMORY LOCATION

DEC ADDR	7A	EXTENDED
DEC disp,X	6A	INDEXED

This instruction decrements the contents of the designated memory location by one. This single instruction replaces the three-instruction sequence of loading an accumulator, decrementing the accumulator, and then storing it back into memory. A similar sequence for incrementing the contents of the memory location is replaced by the increment memory instruction just described. This decrement instruction affects the N, C, and Z flags, and does not affect the C, H, and I flags.

COMPLEMENT THE MEMORY LOCATION

COM ADDR	73	EXTENDED
COM disp,X	63	INDEXED

The contents of the designated memory location are complemented by this instruction. That is, each bit that is initially one is changed to zero, and each bit that is initially zero is changed to one. The N and Z flags are affected by the results of this operation, and the V flag is cleared. The C, H, and I status flags are not affected.

NEGATE THE MEMORY LOCATION

NEG ADDR	70	EXTENDED
NEG disp,X	60	INDEXED

The designated memory location contents are negated, or two's complemented, by the execution of this instruction. As discussed previously, the two's complement is created by subtracting the memory location from zero. The C status flag will be set except when the initial contents of the memory location are zero. The N, V, and Z flags are also affected by this instruction. The H and I flags are not affected.

The following group of instructions allows the programmer to direct the computer to perform arithmetic operations between the designated accumulator and memory location, or between the A and B accumulators. There is also an instruction that, when performed following an addition operation that uses the A accumulator as the operator register, adjusts the A accumulator contents to two BCD digits. The instructions in this group that allow for the four addressing modes require two bytes of memory for the immediate, direct, and indexed modes, and three bytes for the extended mode.

ADD THE MEMORY CONTENTS TO THE ACCUMULATOR

ADDA	#DATA	8B	IMMEDIATE
ADDA	ADDR	9B	DIRECT
ADDA	ADDR	BB	EXTENDED
ADDA	disp,X	AB	INDEXED
ADDB	#DATA	CB	IMMEDIATE
ADDB	ADDR	DB	DIRECT
ADDB	ADDR	FB	EXTENDED
ADDB	disp,X	EB	INDEXED

These instructions add the contents of the designated memory location to the current value in the designated accumulator. The result of the addition is left in the accumulator. All of the status flags, except the I flag, are affected by these instructions. These instructions may be used when it is desired to simply add two eight-bit bytes together. Or, they may be used to add the least significant byte of a multiple precision value without being concerned with the initial condition of the carry flag. In other words, it would not be necessary to clear the carry flag before the addition of the least significant bytes.

ADD THE CONTENTS OF MEMORY PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ADCA	#DATA	89	IMMEDIATE
ADCA	ADDR	99	DIRECT
ADCA	ADDR	B9	EXTENDED
ADCA	disp,X	A9	INDEXED
ADCB	#DATA	C9	IMMEDIATE
ADCB	ADDR	D9	DIRECT
ADCB	ADDR	F9	EXTENDED
ADCB	disp,X	E9	INDEXED

This group of instructions is identical to the previous group except that now the contents of the carry flag are considered as an additional bit to be added to the least significant bit in the accumulator. The results are left in the accumulator. The carry flag is generally used in multiple precision operations to indicate an overflow out of the most significant bit as a result of the addition of the previous pair of bytes. These instructions affect all flags but the I flag.

ADD THE B ACCUMULATOR TO THE A ACCUMULATOR

ABA 1B

This one-byte instruction adds the contents of the B accumulator to the A accumulator. The results of the addition are left in the A accumulator. The condition of all of the status flags except the I flag is affected by the results of this addition.

DECIMAL ADJUST THE A ACCUMULATOR

DAA 19

The decimal adjust accumulator instruction is a one-byte instruction that adjusts the contents of the A accumulator to two binary-coded-decimal digits, one digit in the four least significant bits, and one digit in the four most significant bits. This instruction is used, following the addition of two pairs of BCD digits, to adjust the results to the proper BCD code. Any of the addition instructions just described that leave the result in the A accumulator may be used preceeding this DAA instruction. This instruction operates in the following manner.

The least significant half of the A accumulator is checked for a BCD value of zero to nine, and the H flag is checked for a "zero" condition. If both of these conditions exist, this half is left as is. However, if this half is greater than nine, or the H flag is a "one," the accumulator will be incremented by six, thereby adjusting the least significant half to the proper BCD code. The most significant half is then checked for a BCD value of zero to nine, and the C flag is checked for a "zero" condition. If both conditions are true, the process is complete. Otherwise, if either condition is not met, the most significant half of the A accumulator is incremented by six, with the C flag being set to a "one" if an overflow from the most

significant half occurs. The N and Z flags are also affected by this instruction. The H and I flags are not affected, and the V flag, although it may be changed, has no significance to the result of this operation.

SUBTRACT THE CONTENTS OF MEMORY FROM THE ACCUMULATOR

SUBA	#DATA	80	IMMEDIATE
SUBA	ADDR	90	DIRECT
SUBA	ADDR	B0	EXTENDED
SUBA	disp,X	A0	INDEXED
SUBB	#DATA	C0	IMMEDIATE
SUBB	ADDR	D0	DIRECT
SUBB	ADDR	F0	EXTENDED
SUBB	disp,X	E0	INDEXED

This group of instructions will cause the present value of the designated memory location to be subtracted from the value in the designated accumulator. The result of the subtraction is stored in the accumulator. These instructions affect all the status flags except the H and I flags.

SUBTRACT THE CONTENTS OF MEMORY AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBCA	#DATA	82	IMMEDIATE
SBCA	ADDR	92	DIRECT
SBCA	ADDR	B2	EXTENDED
SBCA	disp,X	A2	INDEXED
SBCB	#DATA	C2	IMMEDIATE
SBCB	ADDR	D2	DIRECT
SBCB	ADDR	F2	EXTENDED
SBCB	disp,X	E2	INDEXED

This group of instructions is identical to the previous group except that now the content of the carry flag is considered as an additional bit to be subtracted from the designated accumulator. This carry flag is generally set or cleared as a result of the subtraction of the previous pair of bytes from two multiple precision values. The carry flag will be set if the previous subtraction required a carry for the subtraction of the most significant bits. The execution of these

instructions leaves the result in the accumulator. All status flags, except the H and I flags, are affected by these instructions.

SUBTRACT THE B ACCUMULATOR FROM THE A ACCUMULATOR

SBA 10

This one-byte instruction subtracts the contents of the B accumulator from the A accumulator. The result is left in the A accumulator. The condition of all status flags, except the H and I flags, is affected by this instruction.

There is a group of instructions that perform a subtraction operation without altering the contents of any CPU registers or memory locations. However, the results of the subtraction operation are indicated by the condition of several of the status flags. The purpose of these instructions is to allow the program to compare the contents of either accumulators, or the index register to a value in memory, or to compare one accumulator to the other, or to test for a zero or negative condition in either of the accumulators or memory.

The following group of compare instructions is a very powerful and somewhat unique set of instructions. They direct the computer to compare the contents of the designated accumulator or index register against the contents of memory, or, in one case, against the other accumulator, and to set the status flags as a result of the compare operation. It is essentially a subtraction operation with the value in the memory being subtracted from the value in the accumulator or index register, except that the value in the accumulator is not actually altered by the operation. However, the flags are set in the same manner as though an actual subtraction operation had occurred. Plus, by subsequently testing the status of the various flags after a compare instruction is executed, the program can determine whether the compare operation resulted in a match or non-match, as the relative magnitude of the two values with respect to each other will be indicated. These various tests are accomplished by utilizing the conditional branch instructions (to be described later).

COMPARE THE CONTENTS OF MEMORY
TO THE ACCUMULATOR

CMPA	#DATA	81	IMMEDIATE
CMPA	ADDR	91	DIRECT
CMPA	ADDR	B1	EXTENDED
CMPA	disp,X	A1	INDEXED
CMPB	# DATA	C1	IMMEDIATE
CMPB	ADDR	D1	DIRECT
CMPB	ADDR	F1	EXTENDED
CMPB	disp,X	E1	INDEXED

This group of compare instructions compares the contents of the designated memory location to the contents of the designated accumulator. These instructions require two bytes for the immediate, direct, and indexed addressing modes, and three bytes for the extended mode. The C, N, V, and Z flags are conditioned according to the results of the subtraction operation, and the H and I flags are not affected.

COMPARE THE B ACCUMULATOR TO THE A ACCUMULATOR

CBA 11

This one-byte instruction compares the contents of the B accumulator to the A accumulator by subtracting the contents of B from A. The flags are conditioned in the same manner as that of the previous compare instruction.

COMPARE THE CONTENTS OF MEMORY
TO THE INDEX REGISTER

CPX	#DATA	8C	IMMEDIATE
CPX	ADDR	9C	DIRECT
CPX	ADDR	BC	EXTENDED
CPX	disp,X	AC	INDEXED

The contents of two successive bytes of memory are compared to the contents of the index register. The memory location (M) is used as the most significant byte, and the succeeding memory location (M+1) is used as the least significant byte of the data to be compared to the contents of the index register. The immediate and

extended addressing modes require three bytes of memory, and the direct and indexed addressing modes require two bytes. Although the N, V, and Z flags are affected by this instruction, only the Z flag may be tested by a conditional branch instruction. The N and V flags do not necessarily give a true indication of the relative magnitude of the two values since they are set up as a result of the subtraction of the most significant halves rather than the entire sixteen-bit values. The C, H, and I flags are not affected.

The following one-byte test instructions "test" the designated register or memory location for a positive, negative, or zero value. This is accomplished by subtracting zero from the designated value. The contents of the designated accumulator or memory location are not altered by this operation. All of these instructions result in the C and V flags being cleared, and the N and Z flags indicating the result of the subtraction. The H and I flags are not affected.

TEST THE ACCUMULATOR

TSTA	4D
TSTB	5D

The contents of the designated accumulator are tested for a positive, negative or zero value. By using the appropriate conditional branch instruction following either of the instructions, the program may perform different operations depending on the results of these test instructions.

TEST THE CONTENTS OF MEMORY

TST	ADDR	7D	EXTENDED
TST	disp,X	6D	INDEXED

These instructions test the contents of the designated memory location for a positive, negative, or zero value, as previously discussed. Here again, the contents of the memory location are not altered by these instructions. The extended addressing mode requires three bytes of memory, and the indexed mode requires two bytes.

There are several groups of instructions that allow Boolean logic operations to be performed between the contents of locations in memory and the accumulator. Boolean logic operations are valuable in a number of programming applications. The 6800 instruction set

allows three basic Boolean operations to be performed. These are the logical AND, logical OR, and EXCLUSIVE OR operations. Each type of logic operation is performed on a bit-by-bit basis between the memory location and accumulator specified by the instruction. A detailed explanation of each type of logic operation, and the appropriate instruction for each type is presented next. These instructions utilize the four basic addressing modes to define the memory location to be used. For this entire group, the immediate, direct, and indexed mode instructions require two bytes of memory, while the extended mode requires three.

"AND" THE ACCUMULATOR

ANDA	#DATA	84	IMMEDIATE
ANDA	ADDR	94	DIRECT
ANDA	ADDR	B4	EXTENDED
ANDA	disp,X	A4	INDEXED
ANDB	#DATA	C4	IMMEDIATE
ANDB	ADDR	D4	DIRECT
ANDB	ADDR	F4	EXTENDED
ANDB	disp,X	E4	INDEXED

When the Boolean AND instruction is executed, each bit of the accumulator will be compared with the corresponding bit in the memory location specified by the instruction. As each bit is compared, a logic result will be placed in the accumulator for each bit comparison. The logic result is determined as follows. If both the bit in the accumulator and the bit in the memory location with which the operation is being performed are a "1," the accumulator bit will be left as a "1." For other possible combinations, (i.e., the accumulator bit = "0" and the memory location bit = "1" or if the accumulator bit = "1" and the memory contents bit = "0" or if both the accumulator and the memory contents have the particular bit = "0"), the accumulator bit will be set to "0." An example will illustrate the logical AND operation:

INITIAL STATE OF THE ACCUMULATOR:	1 0 1 0 1 0 1 0
CONTENTS OF MEMORY LOCATIONS:	1 1 0 0 1 1 0 1
FINAL STATE OF THE ACCUMULATOR:	1 0 0 0 1 0 0 0

The eight logical AND instructions listed above perform this type of logic operation between the designated accumulator and memory location, with the result of the operation stored in the accumulator. The N and Z flags are affected by the results of the logical AND operation, and the V flag is cleared. The C, H, and I flags are not affected.

LOGICAL "OR" THE ACCUMULATOR

ORAA	#DATA	8A	IMMEDIATE
ORAA	ADDR	9A	DIRECT
ORAA	ADDR	BA	EXTENDED
ORAA	disp,X	AA	INDEXED
ORAB	#DATA	CA	IMMEDIATE
ORAB	ADDR	DA	DIRECT
ORAB	ADDR	FA	EXTENDED
ORAB	disp,X	EA	INDEXED

This group of Boolean logic instructions directs the computer to perform the logical OR operation on a bit-by-bit basis with the designated accumulator and contents of the memory location. The logical OR operation will result in the accumulator having a bit set to "1" if either that bit in the accumulator, or the corresponding bit in the memory location is a "1." Since the case where both the accumulator bit and the operand bit is a "1" also satisifies the relationship, that condition will also result in the accumulator bit being a "1." If neither accumulator nor memory location has a "1" in the bit position, the accumulator bit remains "0." An example illustrates the results of the logical OR operation:

INITIAL STATE OF THE ACCUMULATOR:	10101010
CONTENTS OF THE OPERAND REGISTER:	11001101
FINAL STATE OF THE ACCUMULATOR:	11101111

The logical OR instructions listed here perform this operation between the designated accumulator and memory location. The execution of these instructions leaves the result in the accumulator. The effect of these instructions on the status flags is the same as that for the logical AND instructions.

LOGICAL "EXCLUSIVE OR" THE ACCUMULATOR

EORA	#DATA	88	IMMEDIATE
EORA	ADDR	98	DIRECT
EORA	ADDR	B8	EXTENDED
EORA	disp,X	A8	INDEXED
EORB	# DATA	C8	IMMEDIATE
EORB	ADDR	D8	DIRECT
EORB	ADDR	F8	EXTENDED
EORB	disp,X	E8	INDEXED

This group of Boolean logic instructions is a variation of the logic OR. The variation is termed the logical EXCLUSIVE OR. The EXCLUSIVE OR operation is similar to the OR, except that when the corresponding bits in both the accumulator and the operand register are a "1," the accumulator bit will be set to "0." Thus, the accumulator bit will be a "1" after the operation only if just one of the registers (accumulator or memory location) has a "1" in the bit position. (Again, the operation is performed on a bit-by-bit basis.) An example provides clarification:

INITIAL STATE OF THE ACCUMULATOR:	1 0 1 0 1 0 1 0
CONTENTS OF THE OPERAND REGISTER:	1 1 0 0 1 1 0 1
FINAL STATE OF THE ACCUMULATOR:	0 1 1 0 0 1 1 1

These logical EXCLUSIVE OR instructions, similar to the AND and OR instructions, perform the operation between the designated accumulator and memory location with the results of the operation being stored in the accumulator. The status flags are also affected, or not affected, in the same manner as the logical AND instructions.

BIT TEST THE ACCUMULATOR

BITA	# DATA	85	IMMEDIATE
BITA	ADDR	95	DIRECT
BITA	ADDR	B5	EXTENDED
BITA	disp,X	A5	INDEXED
BITB	# DATA	C5	IMMEDIATE
BITB	ADDR	D5	DIRECT
BITB	ADDR	F5	EXTENDED
BITB	disp,X	E5	INDEXED

Along the same lines as the compare and test instructions, which perform arithmetic operations without altering the contents of the accumulator or memory location, this BIT TEST instruction performs a logic AND between the designated accumulator and memory location without altering either of their contents. However, the result of the operation causes the status flags to be affected in the same manner as for the logical AND instruction. This instruction is useful when it is desired to determine whether an individual bit or group of bits in a designated memory location are set to one or zero. One could load the desired accumulator with zeros in all bits except for the bit location to be tested. These bits would be set to one. Then, by executing this BIT instruction with the addressing mode indicating the desired memory location, the Z flag would indicate whether any of the designated bits are one. Or, if the most significant bit is being tested, its condition would be indicated by the condition of the N flag.

The 6800 has a group of instructions that allows the programmer to condition several of the status flags individually, or condition all of the flags by loading the flag register with the contents of the A accumulator. All of the instructions in this group require only one byte of memory. The instructions in this group that refer to an individual flag affect only that flag. The other status flags remain in their initial condition.

SET THE CARRY FLAG

SEC 0D

This instruction sets the carry flag to a value of "1." This instruction, and the "clear the carry" instruction presented next, provides a convenient method for conditioning the carry flag before an arithmetic or rotate instruction.

CLEAR THE CARRY FLAG

CLC 0C

This instruction clears the carry status flag by loading it with a "0."

SET THE INTERRUPT FLAG

SEI 0F

The interrupt flag is set to a "1" by the execution of this instruction. This instruction may be considered a disable interrupt instruction, since the interrupt flag disables the CPU from accepting maskable interrupts when it is set to a "1."

CLEAR THE INTERRUPT FLAG

CLI 0E

This instruction clears the interrupt flag to a "0" condition. Clearing the interrupt flag allows the CPU to accept interrupts from the maskable interrupt line.

This pair of interrupt flag instructions provides the programmer with a means of determining when the computer may accept interrupts on the maskable interrupt line. The function of these two instructions is automatically performed when an interrupt is received. That is, when any of the three possible interrupts are received, the computer automatically sets this I flag. Then, upon execution of the "return from interrupt" instruction (to be presented later), the I flag is returned to its initial state at the time the interrupt was received. There may also be times in a program when an operation to be performed affects data critical to the execution of the interrupt service routine. Before performing this operation, the interrupt flag should be set so that a maskable interrupt will not be accepted while the data is being changed. Once the program has completed this operation, the I flag may be cleared to allow interrupts to be received.

SET THE OVERFLOW FLAG

SEV 0B

This instruction sets the two's complement overflow flag to a "1." This instruction, and the following "clear the overflow flag" instruction, provides a convenient method for controlling the V flag.

CLEAR THE OVERFLOW FLAG

CLV 0A

This instruction clears the two's complement overflow flag to a "0."

TRANSFER THE A ACCUMULATOR TO THE STATUS FLAG REGISTER

TAP 06

This instruction transfers the contents of the A accumulator to the status flag register. The status flag register is an eight-bit register that has one bit assigned for each of the six flags, and the two most significant bits always set to one. (This register is defined in detail at the beginning of this chapter.) Obviously, all of the status flags are affected by this instruction.

TRANSFER THE STATUS FLAG REGISTER TO THE A ACCUMULATOR

TPA 07

This instruction performs the reverse operation of the previously defined instruction. It transfers the contents of the status flag register to the A accumulator. This would allow the program to then store the status in a memory location for future reference, should it be desired to refer to it at some future point in the program. The condition of the flags is not affected by this instruction.

It is often desirable to be able to shift the contents of an accumulator or memory location either right or left. In a fixed length register, a simple shift operation would result in some information being lost because what was in the MSB (most significant bit) or LSB (least significant bit), depending on which direction the shift occurred, would be shifted right out of the register! Therefore, instead of losing this information, the carry flag is used as an extension of the accumulator or memory location to "catch" the bit being shifted out of either the LSB for a shift to the right, or the MSB for a shift to the left.

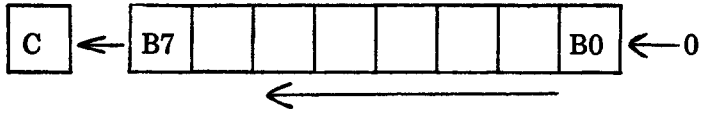
When performing these shift operations, the condition of the bit being shifted into the register must also be considered. Depending on the application of the shifting operation, it may be desired to shift a zero or a one into this bit, or to shift the initial contents of the carry

flag into this bit. In some cases, one may want to maintain this bit in its initial state, which may be accomplished by shifting it back into itself and at the same time shifting it into the adjacent bit. The shifting operation that shifts the carry around to the opposite end of the register is termed a "rotate" operation, since the initial contents of the entire register and the carry are never lost. It is shifted out one end into the carry, and from the carry back into the other end of the register.

The 6800 CPU provides five various shifting and rotating operations that may use either the A or B accumulators, or a memory location as the register to be shifted. A graphic and verbal description of the shift and rotate operations available are presented next. For each of the shift and rotate instructions described here, those designating an accumulator require one byte, those using the indexed addressing mode require two bytes, and those indicating the extended addressing mode require three bytes.

ARITHMETIC SHIFT LEFT

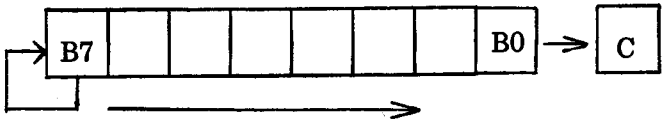
ASLA	48	
ASLB	58	
ASL ADDR	78	EXTENDED
ASL disp,X	68	INDEXED



The arithmetic shift left operation shifts either the designated accumulator or memory location to the left one bit, with the MSB shifted into the carry and a zero shifted into the LSB. This operation essentially multiplies the initial contents of the register by two. For multiple precision operations, this shift instruction may be used to shift the least significant byte, and the successive bytes may be shifted by using the rotate left instruction, to be described shortly. By starting with this instruction, it would not be necessary to initially clear the carry flag. The C, N, and Z flags are affected by this operation. The V flag will be set if the C and N flags are in opposite states after the execution of this instruction. Otherwise, the V flag will be cleared. The H and I flags are not affected.

ARITHMETIC SHIFT RIGHT

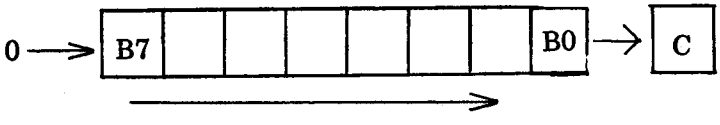
ASRA	47	
ASRB	57	
ASR ADDR	77	EXTENDED
ASR disp,X	67	INDEXED



The arithmetic shift right operation shifts the contents of the designated accumulator or memory location one bit to the right with the least significant bit shifted into the carry, and the most significant bit shifted into bit six and back into itself. This maintains the sign bit of the register, which is required for two's complement arithmetic operations. For multiple precision operation, this instruction may be used to shift the most significant byte to the right, and the subsequent bytes may be shifted by using the rotate right instruction, to be described shortly. The effect on the status flags is the same as that for the arithmetic shift left.

LOGICAL SHIFT RIGHT

LSRA	44	
LSRB	54	
LSR ADDR	74	EXTENDED
LSR disp,X	64	INDEXED

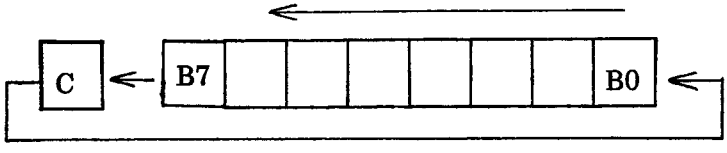


The logical shift right instruction shifts the designated register to the right one bit. Bit zero is loaded into the carry flag, and bit seven is loaded with a zero. This instruction is used to divide the contents

of the register by two when the MSB is assumed to be part of the value and not the sign of the value. The C, N, and Z flags are affected by the result of this operation. The V flag will take on the same conditions as the C flag, and the H and I flags are not affected.

ROTATE LEFT

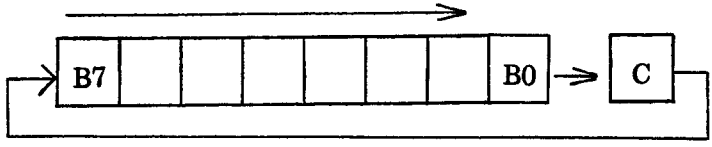
ROLA	49	
ROLB	59	
ROL ADDR	79	EXTENDED
ROL disp,X	69	INDEXED



The designated accumulator or memory location is rotated one bit to the left by the execution of this instruction. The MSB is rotated into the carry, and the initial contents of the carry are rotated into the LSB. Since this instruction forms a closed loop, it does not lose the contents of any of the bits. It may, therefore, be used to calculate the parity of the value in the register by rotating each bit into the carry and adding up the number of ones contained in the register. Rotating a multiple precision value to the left may be accomplished by initially clearing the carry and then, beginning with the least significant byte, rotating each byte once to the left. In doing so, it is essential that the instructions in between each rotate do not affect the carry flag. The status flags are affected in the same manner as that for the arithmetic shift left.

ROTATE RIGHT

RORA	46	
RORB	56	
ROR ADDR	76	EXTENDED
ROR disp,X	66	INDEXED



The rotate right instruction rotates the designated accumulator or memory location once to the right with the LSB rotated into the carry, and the initial contents of the carry rotated into the MSB. The parity of the contents of the register may also be checked by a series of rotate right instructions. Dividing a multiple precision value by two may be accomplished by initially clearing the carry and then, beginning with the most significant byte and working down to the least significant byte, each byte of the multiple precision value is rotated once to the right. Here again, the instructions in between rotate instructions must not affect the carry. The status flags are affected in the same manner as that for the arithmetic shift left instruction.

THE NO OPERATION INSTRUCTION

NOP 01

The no operation, or NOP, instruction directs the computer to consume time by executing a machine cycle that effectively does nothing except advance the program counter to the next memory address! None of the CPU registers are affected by the operation. The instruction is useful when creating time delays, or as a filler if patches to a program are required (or anticipated).

The instructions discussed so far have been sort of direct action instructions. The programmer arranges a sequence of these types of instructions in memory, and when the program is started, the computer proceeds to execute the instructions in the order in which they are encountered. The computer automatically reads the contents of the memory location, executes the instruction it finds there, and then automatically increments a special address register called a "program counter" that will result in the machine reading the information contained in the next sequential memory location. However, it is often desirable to perform a series of instructions located in one section of memory, and then skip over a group of memory locations to start executing instructions in another section of memory. This

action can be accomplished by a group of instructions that will cause a new address value to be placed in the program counter. This will cause the computer to go to a new section of memory and continue executing instructions sequentially from the new memory location.

The branch instructions in this computer add considerable power to the machine's capabilities because there are a series of conditional branch instructions available. That is, the computer can be directed to test the status of a particular flag or group of flags, and if the status of the flag(s) is the desired one, a branch will be performed. If it is not, the computer will continue to execute the next instruction in the current sequence. This capability provides a means for the computer to make decisions, and to modify its operation as a function of the status of the various flags at the time that the program is being executed.

All of the branch instructions use the relative addressing mode to define the memory location from which the next instruction to be executed is to be taken. This mode of addressing requires two bytes of memory to properly define the instruction. The first byte contains the machine code for the type of branch instruction to be executed, and the second byte contains the relative displacement, in two's complement form, from the memory location following the second byte of the branch instruction. (One may wish to refer back to the beginning of this chapter at this time, to review the discussion of the relative addressing mode.) A description of each of the branch instructions is presented next. It should be noted that these branch instructions do not affect any of the status flags.

UNCONDITIONAL BRANCH

BRA RELA 20

The unconditional branch instruction always branches to the relative address designated in the second byte of the instruction. It does not test the condition of any of the status flags before determining whether to branch or not.

The following list of branch instructions tests a single flag to determine whether to branch, or to fall through to the next sequential instruction. The first column contains the mnemonic representation for the instruction, the second column contains the machine code for

the first byte of the instruction, and the final column indicates the flag tested and the condition that would cause the instruction to branch.

BCC	RELA	24	C = 0
BCS	RELA	25	C = 1
BNE	RELA	26	Z = 0
BEQ	RELA	27	Z = 1
BPL	RELA	2A	N = 0
BMI	RELA	2B	N = 1
BVC	RELA	28	V = 0
BVS	RELA	29	V = 1

The following pair of branch instructions is generally used following a compare or subtract operation to indicate the relative magnitude between the minuend and subtrahend values. These two instructions assume that the two eight-bit bytes being compared are unsigned binary values; the MSB is NOT considered a sign bit. In other words, a hexadecimal value of FF is greater than zero.

BRANCH IF HIGHER

BHI	RELA	22	C = 0 and Z = 0

This instruction will branch to the relative address if a preceeding compare or subtract instruction indicates that the contents of the accumulator are greater than the contents of the operand. This condition is indicated by both the C and Z flags being equal to zero.

BRANCH IF LOWER OR SAME

BLS	RELA	23	C = 1 and Z = 1

This instruction will branch to the designated relative address if, following a compare or subtract instruction, the contents of the accumulator are indicated to be less than or equal to the contents of the operand. This condition is indicated by both the C and Z flags being equal to one.

This final group of conditional branch instructions is generally used following a compare or subtract operation in which it is assumed the two values operated on are two's complement binary

values. That is, the MSB of each value is assumed to indicate the sign of the value. The hexadecimal value of FF is considered to be less than zero.

BRANCH IF GREATER THAN

BGT	RELA	2E	Z = 0 and N = V

This instruction branches to the relative location if the two's complement value in the accumulator is greater than the two's complement value in the operand. This condition is indicated by the Z flag being equal to zero, and the N flag being equal to the V flag.

BRANCH IF GREATER THAN OR EQUAL

BGE	RELA	2C	N = V

This instruction branches to the relative location if the two's complement value contained in the accumulator is greater than or equal to the two's complement value contained in the operand, indicated by the N flag being equal to the V flag following a compare or subtraction operation.

BRANCH IF LESS THAN OR EQUAL

BLE	RELA	2F	Z = 1 or N ≠ V

This instruction branches to the relative address if a compare or subtract operation indicates that the two's complement value contained in the accumulator is less than or equal to the two's complement value contained in the operand. This condition is indicated by either the Z flag being set, or the N flag being in the opposite condition of the V flag.

BRANCH IF LESS THAN

BLT	RELA	2D	N ≠ V

This instruction branches to the relative location if the compare or subtract operation indicates that the two's complement value contained in the accumulator is less than the two's complement value contained in the operand. This condition is indicated when, follow-

ing the compare or subtract operation, the N flag is in the opposite condition of the V flag.

THE JUMP INSTRUCTION

JMP	ADDR	7E	EXTENDED
JMP	disp,X	6E	INDEXED

The jump instruction is similar to the unconditional branch instruction in that it always results in the computer going to the designated address rather than fetching the next instruction from the current sequence. However, the jump instruction is not limited to an area in memory relative to its current location. Using either of the addressing modes indicated, the jump instruction can direct the computer to any location throughout its memory. For the three-byte extended addressing mode instruction, the second byte contains the page portion, and the third byte contains the low portion of the address that the computer is to jump to. For the indexed mode, the address is obtained by adding the indicated displacement to the contents of the index register at the time the instruction is executed. As with the branch instructions, this jump instruction does not affect any of the status flags.

Quite often, when a programmer is developing computer programs, he will find that a particular algorithm (sequence of instructions for performing a function) can be used many times in different parts of the program. Rather than having to keep entering the same sequence of instructions at different locations in memory (which would not only consume the time of the programmer but would also result in a lot of memory being used to perform one particular function), it is desirable to be able to put an often used sequence of commands in one location in memory. Then, whenever this particular algorithm is required by another part of the program, it would be convenient to jump to the section that contained the often-used algorithm, perform the sequence of instructions, and then return back to the main part of the program. This is a standard practice in computer operation. The frequently used algorithm can be designated as a subroutine. A special set of instructions allows the programmer to "call" - in other words, specify a special type of jump to - a subroutine. A second type of instruction is used to terminate a sequence of instructions that is to be considered a subroutine. This special terminator will cause the program operation to revert back to

the next sequential location in memory following the instruction that called the subroutine. The 6800 provides both a branch and jump to subroutine instruction. A single return from subroutine instruction is provided to terminate a subroutine.

When a jump or branch to subroutine instruction is executed, the CPU will save the address of the instruction following the subroutine call, to be used to return from the subroutine, by storing it in the stack. The address in the program counter is advanced to the location following the subroutine call instruction. The low portion of this address is then stored in the stack indicated by the stack pointer. The stack pointer is then decremented by one, and the page portion of the address is then stored in the stack. Finally, the stack pointer is decremented once more to position it for the next stack operation.

The return instruction that terminates a subroutine only requires one byte. When the CPU encounters a return instruction, it causes the address stored in the stack to be pulled off the stack into the program counter, thus returning the execution of the program to the calling routine. The page, and then low, portions of the address are each pulled from the stack in the same manner that a value is pulled from the stack and loaded into an accumulator.

BRANCH TO SUBROUTINE

BSR	RELA	8D

This two-byte instruction uses the relative addressing mode to determine the start location of the subroutine to be entered. The address following the second byte of this instruction is stored in the stack as the return address for the subroutine call. None of the status flags are affected by this instruction.

JUMP TO SUBROUTINE

JSR	ADDR	BD	EXTENDED
JSR	disp,X	AD	INDEXED

This instruction directs program execution to the address indicated by the addressing mode. For the three-byte extended addressing mode instruction, the second byte contains the page portion, and the third byte contains the low portion of the start address of

the subroutine to be called. The address of the byte following the third byte of this instruction is stored in the stack as the return address. For the two-byte indexed mode, the start address of the subroutine is derived by adding the displacement contained in the second byte of the instruction to the contents of the index register. The memory location following the second byte of this instruction is stored in the stack as the return address. These instructions do not affect the status flags.

RETURN FROM SUBROUTINE

RTS 39

This one-byte instruction pulls the next address from the stack and places it in the program counter. Program execution will continue with the instruction at this address. None of the status flags are affected.

This final group of instructions deals with the software portion of interrupt operations. These instructions, along with the interrupt flag set and clear instructions presented previously, provide the 6800 with the necessary software capability to operate under interrupt control.

WAIT FOR INTERRUPT

WAI 3E

When an interrupt is received, the computer suspends operation of the program currently being executed and stores the contents of the CPU registers in the stack. This register storage operation requires a minimum of nine microseconds to be performed. In some cases, this delay before actually servicing the interrupt may not be desired. Therefore, to overcome this problem, this "wait for interrupt" instruction may be used to store the registers before the interrupt is received, thereby eliminating this delay. Following the execution of this instruction, program operation is suspended until an interrupt is received. When the interrupt is received, the interrupt flag is set and the CPU immediately "vectors" to the interrupt service routine. After the interrupt service routine has completed its operation, it will return execution to the instruction following the wait for interrupt instruction. The execution of this one-byte instruction does not affect any of the status flags.

SOFTWARE INTERRUPT

SWI 3F

The execution of this one-byte instruction causes the 6800 to respond in a similar manner to that when a hardware interrupt is received. The current contents of the CPU registers are stored in the stack, and the address in the software interrupt vector is used to select the next instruction to be executed. Also, the interrupt flag is set, disabling interrupts on the maskable interrupt line. The memory location immediately following this instruction must contain the first instruction to be executed upon returning from the software interrupt service routine.

RETURN FROM INTERRUPT

RTI 3B

This one-byte instruction is used at the completion of an interrupt service routine to automatically restore the contents of the CPU registers to their initial values at the time the interrupt was received. This is accomplished by pulling them from the stack in the reverse order in which they were stored. In doing so, all of the status flags are also restored to their initial state at the time of the interrupt.

INFORMATION ON INSTRUCTION EXECUTION TIMES

When programming for real-time applications, it is important to know how much time each type of instruction requires to be executed. With this information, the programmer can develop timing loops or determine with substantial accuracy how much time it takes to perform a particular series of instructions. This information is especially important when dealing with programs that control the operation of external devices that require events to occur at specific times. Along with the list of mnemonics and machine codes, Appendix A provides the nominal instruction execution time for each instruction used in a 6800 system. The table shows the number of cycle states required by the instruction. Since the nominal cycle time for a 6800 microcomputer is one microsecond, the number of cycle states translates directly into the execution time for each instruction. In

some cases, however, the cycle time of a 6800 system may be slowed down to allow the use of slower memory. To calculate the execution time of an instruction, multiply the number of cycle states for the instruction by the time required for one cycle. Knowing the exact time required by the CPU to execute each instruction allows algorithms to be programmed to have specific events occur at precisely timed intervals. This concept is discussed in greater detail in Chapter three on programmed time delays.

6800 PROGRAMMING TECHNIQUES

A number of different programming techniques and options are available to the programmer when developing programs for the 6800. These various techniques are made possible by the structure of the 6800 instruction set. The different methods provided for storing and retrieving data, and altering the instruction execution sequence provide the programmer with a great deal of flexibility. Proper selection and utilization of these techniques can shorten both memory requirements and execution time for a given program.

Although the 6800 is capable of directly addressing 64K of memory, the instruction set places special significance on the lowest 256 bytes. This portion of memory is referred to as page zero. The reason for the significance of using page zero is the presence of the direct addressing mode.

As one should be aware by now, the direct addressing mode allows one to directly reference a location on page zero by specifying only the eight least significant bits of the address, rather than an entire sixteen bit address. The importance of this addressing mode lies in the fact that only one byte is required to specify the memory address rather than two bytes, as in the extended mode. For example, an instruction to store the contents of the A accumulator in location 10 on page 00 requires only two bytes - one byte to indicate that the instruction is a "store A direct" instruction, machine code 97, and the second byte to indicate the location on page zero in which the data is to be stored, 10. If the same data were to be stored in location 10 on page 01, the extended mode version of the STAA instruction would be required. This extended mode instruction would use three memory locations - one location for the machine code, B7, and two more to designate the address, 01 10. This one byte difference between the two modes may not appear to be a significant savings in memory; however, it does add up as the length of the program increases.

Another method for shortening a program using the direct addressing mode is to store a table of addresses on page zero that are frequently loaded into the index register. For example, suppose the index register is loaded with the address 0300 at numerous points throughout a program. This address may be the start address of a conversion table or table of data required by a program. The novice's first instinct is generally to use a "load the index register

immediate" instruction each time the index register is to be set to this address. This instruction requires three bytes of memory - one byte for the machine code, CE, and two bytes to designate the address, 0300. This method is satisfactory if the address is only loaded once or twice throughout the program. However, if it is loaded three or more times, one could conserve memory by storing the address in a pair of memory bytes on page zero, and use the two byte direct addressing mode instruction to load the index register. The diagram below illustrates how the address stored at location 40 on page 00 is loaded into the index register using the two byte direct mode instruction.

Before Execution Of LDX $40 (Machine Code DE 40):

MEMORY LOCATION	CONTENTS	INDEX REGISTER
0040	03	0000
0041	00	

After Execution of LDX $40 (Machine Code DE 40):

MEMORY LOCATION	CONTENTS	INDEX REGISTER
0040	03	0300
0041	00	

An alternative to using page zero for temporary data is to set up the data wherever it may be convenient, and utilize the index register to point to it. Just as the direct mode instructions require two bytes of memory, the indexed mode instructions also require two bytes. The first byte indicates the machine code for the instruction to be executed, and the second byte indicates a forward displacement from the current value contained in the index register. Therefore, using the indexed mode one may set up a table area anywhere in memory, load the index register with the lowest address of the table area, and, using the two byte indexed mode instructions, access any location from that indicated by the index register up through the index register plus FF. That is, once the index register is set to a specific memory location, any memory location within the forward displacement limit of FF may be directly accessed by a two byte instruction.

The index register may also be used to point to consecutive memory locations for storing and retrieving data. These memory locations may be used to store characters as they are received from an input device, or a character string to be output to a display device. Or, they may contain a multiple precision value that is to be manipulated by the program. By setting the index register to the first location, the contents of that location may be manipulated by using the two byte indexed addressing mode instructions. After the operation on the first byte has been completed, the index register may be incremented by the INX instruction to advance it to the next memory location, and the program may loop back to perform the same instruction sequence on the next memory location. By creating a program loop in this manner, consecutive memory locations may be operated on by the same instruction sequence.

The program listing below illustrates this program loop method for outputting a string of characters. Before entering this routine, the index register must be set to the start address of the message to be outputted. Each character is loaded into the A accumulator. The character is then tested to determine whether it is an all zero byte. This code indicates that the end of the character string has been reached, and the program loop is completed. When this occurs, the program branches out of the loop, to the label ZCHAR which contains a return instruction. Setting up the program in this manner allows this message routine to be used as a common subroutine for outputting different character strings stored in memory. If the terminating character is not detected, a routine referred to as ECHO is called to output the character to the display device. Upon returning from the ECHO routine, the index register is incremented to point to the next character in the string and the program loops back to process the next character. (The ECHO routine would be an output driver for a printer or display device used on one's system. One may refer to the chapter on I/O processing for suggestions for this routine.)

```
MSSAGE  LDAA X      Fetch character from character string
        BEQ ZCHAR   Character = 0? Yes, end of message
        JSR ECHO    No, output character to display device
        INX         Advance message pointer
        BRA MSSAGE  Loop back to process next character
ZCHAR   RTS         Message output complete, return
```

Another means of detecting when the program loop has completed its operation is to set up an accumulator or memory location as a countdown register. This register would initially contain a binary count of the number of times the program loop is to be executed. Then, for each pass through the loop, the countdown register would be decremented and checked for a value of zero. If the register did not go to zero, the program would remain in the loop. When the register does reach zero, the program would jump out of the loop. The following program listing performs the same operation as the MSSAGE routine, however, rather than check for the terminating zero byte, the program uses the B accumulator as a countdown register. Before calling this routine, the index register would be set to the start location of the message to be printed, and the B accumulator would be set to the number of characters to be outputted.

```
MESAG   LDAA X      Fetch character from character string
        JSR ECHO    Output character to display device
        INX         Advance message pointer
        DECB        Decrement countdown register
        BNE MESAG   Countdown =0? No, loop back
        RTS         Yes, message output complete, return
```

It is often desired to have a computer count the number of times a specific operation is performed. The counter may be indicating the number of times a specific routine has been executed or the number of characters received from an input device. There are several ways such a counter may be set up. One way is to designate one of the accumulators as the counter register, and each time a count is to be made, the INCA or INCB instruction may be executed. Since the accumulators are eight bits long, they may be used to indicate a count from 0 to 255. If a greater number of occurances is expected, the sixteen bit index register may be used as the counter. Sixteen bits would allow for a maximum count of 65,535. To set up either an accumulator or the index register as a counter, one would simply load the designated register with all zeros and insert the proper increment instruction at the location within the program where the count is to be made. When the process is complete, the designated accumulator or index register will contain a binary count of the number of times the operation occurred.

The accumulator and index register are generally required to per-

form other operations while a counter is to be maintained. In this case, one or more memory locations may be designated as the counter storage area. Once again, if the count is expected to exceed 255, several memory locations would have to be used for the counter. For a single memory location, an increment instruction using the extended addressing mode would be inserted in the instruction sequence where the count is to be made. When two or more memory locations are to be used for the counter, a series of instructions would be needed to increment the counter. The instruction sequence is that of incrementing the memory location containing the least significant portion of the count, and then checking its contents for a value of zero. If the least significant portion went to zero as a result of the increment, the next memory location would also be incremented. If a third byte of memory is used as part of the counter, its contents would be incremented whenever incrementing the second byte resulted in the second byte going to zero. A sample program listing for incrementing a triple precision counter is presented next. The memory locations used for the counter are designated CNTR1, CNTR2, and CNTR3.

```
COUNTR  INC CNTR1     Increment the LSByte of the counter
        BNE OVER      =0? No, skip other incr instruction
        INC CNTR2     Yes, increment 2nd byte of counter
        BNE OVER      =0? No, skip next instruction
        INC CNTR3     Yes, incr the MSByte of the counter
OVER    ...           Continue processing
```

One programming element that can aid in creating effective programs is the stack pointer. The reason for its effectiveness is that a number of data storage and retrieval operations may be performed by single byte instructions. These instructions operate with either the contents of a single accumulator, or the contents of all of the CPU registers and status flags.

The instructions that work on the contents of an individual accumulator are the PUSH and PULL instructions. These instructions allow one to transfer data to and from the stack with the A and B accumulators. As discussed in chapter one, the PSHA and PSHB instructions store the contents of the respective accumulators in the stack at the location indicated by the stack pointer. Following the storage, the stack pointer is automatically decremented to position

it properly for the next stack operation. The PULA and PULB instructions cause the stack pointer to be incremented, and the data in the stack indicated by the stack pointer is then transferred to the designated accumulator. These instructions are especially useful for temporarily saving the contents of an accumulator when the accumulator is needed to perform another operation. When the operation is complete, the data stored in the stack may be retrieved by executing a PULL instruction. Storing data in this manner not only saves an extra byte or two for each of the push and pull instructions, it also frees the programmer from setting up specific storage areas for saving this data. (It should be noted that data pushed from one accumulator onto the stack may be pulled off the stack and into either accumulator. It is not necessary to use the same accumulator when recalling the data, unless the routine dictates it.)

During the process of writing programs, one occasionally encounters a condition in which it would be desirable to save the contents of all of the CPU registers, perform some other operation, and then restore the initial contents of the CPU registers and continue with the original processing. For this type of operation, the 6800 provides the software interrupt instruction (mnemonic SWI), and the return from interrupt instruction (mnemonic RTI).

The SWI instruction may be thought of as a subroutine call that stores all registers and status flags before entering the subroutine. The registers and status flags are pushed onto the stack in the following order: program counter, index register, accumulator A, accumulator B, and condition code register. The CPU then fetches the start address for the software interrupt routine from the third interrupt vector, which is assigned to the software interrupt. The programmer must insert the software interrupt routine at the location specified by this vector. Since all of the registers have been saved before entering this routine, the software interrupt routine is free to use the accumulators and index register as it may require.

Since the SWI instruction operates as an interrupt, the I flag is set by its execution. If, during the software interrupt routine, it is desired to allow the acknowledgment of maskable interrupts, the software interrupt routine should clear the I flag when it is entered.

Once the software interrupt routine completes its operation, it returns to the original program by executing the RTI instruction. This instruction restores the CPU registers and status flags to their

initial contents at the time the software interrupt instruction was executed. Thus, the original program may continue execution without using up program steps to restore the register contents.

For systems that are designed for use with the MIKBUG** program, the software interrupt vector is set to a specific routine within the MIKBUG** program. For systems that use the MIKBUG** program, the capability to set up a software interrupt subroutine is most likely not available to the user. However, a very useful function is programmed into the MIKBUG** program for using the SWI instruction. This function is that of setting a breakpoint within a program being debugged.

A breakpoint is a location within a program at which the programmer desires to examine the progress of the program execution. The purpose of a breakpoint routine is to stop the execution of the program at the breakpoint location and save the contents of the CPU registers and status flags. The programmer may then examine the registers, the condition of the status flags, and the contents of any data storage areas within memory. By examining these values, the programmer can determine whether the program is operating as is expected up to the breakpoint.

Setting a breakpoint for the MIKBUG** program is accomplished by simply storing the machine code for the SWI instruction (3F) in the first byte of the instruction at which program execution is to be examined. One must insure that this code is stored in the first byte of a two or three byte instruction, otherwise the result will only be that of altering the execution of the program being tested. After the breakpoint is set, program execution may begin. When the program reaches the breakpoint, the CPU registers and status flags are stored, and the MIKBUG** control routine is entered. The programmer may then examine the register values and memory contents using the MIKBUG** commands. If the program appears to be operating correctly up to that point, the instruction at the breakpoint may be restored, and another breakpoint may be set further into the program. Then, by entering a GOTO command, the program will resume execution at the first breakpoint location until a second breakpoint is encountered. This allows one to step through programs from one breakpoint to another and examine the program's operation at each point.

The proper operation of a program is often dependent on being located at a specific area in memory. This requirement is created by the use of jump and jump to subroutine instructions that call out specific memory locations for their destinations. If this type of program is to be moved to a different location in memory, it would be necessary to revise the machine code of the program to account for the new destination addresses of the jump and jump to subroutine instructions. There are times, however, when it is desirable to have the capability to locate a routine at any area in memory without having to reassemble each time it is moved. This type of program is referred to as a relocatable program.

A relocatable program is one that may be loaded into any continuous block of memory without altering the exact machine code. This type of program is developed when a routine is written that is to be used by numerous other programs, and it is not desired to reassemble it each time it is to be incorporated in a different program. The relocatable routine may be placed in any area in memory that is convenient for the program using it.

The branch instructions of the 6800 provide an easy means of creating relocatable programs. Since the destination of the branch instructions is specified as a displacement relative to the location of the branch instruction, it is not necessary to specify the destination of the branch as an exact memory location. Instead, the destination is represented as a forward or backward displacement from the location of the branch instruction plus two. As discussed in chapter one, this displacement has a limit of -128 to +127 (decimal) locations from the branch instruction plus two.

Writing a relocatable program that will reside in less than one page of memory can usually be accomplished by simply changing any jump or jump to subroutine instructions to their branch or branch to subroutine counterparts. If a branch destination is not within the allowable limits, it may be possible to rearrange the various portions of the routine to move the destination within the required limits. However, if it is not possible to rearrange the routine, or if the routine takes up more than one page of memory, it is necessary to insert extra branch instructions to link the original branch to its proper destination. These intermediate branch instructions shall be referred to as ISLANDS.

The ISLANDS in a relocatable program create a type of stepping

stone for a branch that must exceed its displacement limit. For example, if the destination of a branch instruction is located 200 (decimal) locations ahead of the branch, it would be necessary to insert an island approximately halfway between the branch instruction and its destination. Since a branch has a forward displacement of 127 (decimal), the island should be inserted approximately 112 (decimal) or 70 (hexadecimal) locations after the branch instruction. This places it within the limits of the original branch and within its limits of the desired destination. If the original branch and destination are separated by more than 256 locations, it would be necessary to insert more than one island. In this case, the original branch instruction would go to one island, which would branch to a second island, which would in turn branch to the third island, and so on until the destination is within the limits of the final island.

Along with falling within the limits of the branch instructions, the placement of the island must be such that the logic of the program is not disrupted. One cannot indiscriminately locate an island in the middle of an instruction sequence, as this would alter the logic of the program. One should look for the end of an instruction sequence, designated by an unconditional branch or jump or return from subroutine instruction. An island may be inserted immediately following either of these instructions, since they represent the point at which the sequential execution of the program instructions is terminated. If there is no BRA, JMP or RTS instruction within a reasonable distance of the desired location for the island, one may insert a second branch instruction along with the island. This additional branch instruction must be placed before the island, and have as its destination the instructions immediately following the island. This allows the instruction sequence to branch around the island and continue with normal execution. An example of this method of inserting an island is given next.

Original Instruction Sequence

```
...
RORA        Rotate A right
ABA         Add B to A
NEGA        Negate A
...
```

After Island Insert

```
            . . .
            RORA        Rotate A right
            ABA         Add B to A
            BRA OVERIS  Branch over island
            BRA ILND1   Island insert for relocatable program
OVERIS      NEGA        Negate A (logic sequence unaffected)
            . . .
```

Several of the programming techniques discussed in this chapter are used by various routines throughout the remainder of this book. In particular, Appendix F contains a relocatable version of the floating point routines presented in Chapter 5. As one develops their own routines and programs for their 6800 system, one should try to corporate the techniques described here to make their programs as efficient as possible.

GENERAL PURPOSE ROUTINES

Whenever one writes a program, there are usually several basic operations that are included in the program in the form of subroutines. Although the overall objective of various programs may be to perform totally unrelated tasks, the same group of instructions may be called on to provide these basic functions. These subroutines perform such functions as clearing a section of memory, moving a block of data from one section of memory to another, manipulating multiple precision numbers, comparing data, creating software delays, generating random numbers, and checking parity.

This chapter contains a number of routines that may be used to aid in fulfilling the subroutine requirements of many programs. These routines may be set up as subroutines, as presented here, if their operation is required by several portions of a program. Or, they may be revised to be used in line with the main instruction sequence of a program, should their operation be required only once by the program. Such revision requires either eliminating the return instruction at the end of the routine, or changing the return instruction within the routine to a branch or jump instruction.

The following list is a summary of the subroutines presented in this chapter.

Clearing a section of memory
Transferring a section of memory
Operations with multiple precision values
- Incrementing a multiple precision value
- Decrementing a multiple precision value
- Rotating a multiple precision value
- Complementing a multiple precision value
- Multiple precision addition and subtraction
- Comparing two multiple precision values

Checking for a value within given limits
Programmed time delays
Random number generators
Checking parity

The subroutines presented in this chapter make use of the index register as the pointer to the data being operated on. Also, whenever a counter is required, the B accumulator is used. Obviously, with the numerous addressing modes available, there are numerous possible variations of the routines presented. When applying any of these routines to a specific program, one should consider alternate addressing modes which may better suit the application. Although these alternates are not specifically presented, the general operation of the routine is discussed, and the reader should have little trouble in making the conversion.

CLEARING A SECTION OF MEMORY

When setting up a program for entering data or storing the results of a calculation in a section of memory, it is often desirable to clear the memory locations to be used for the storage. This operation is achieved by filling the memory locations with all zeros. One way to do this would be to perform a series of CLR ADDR instructions in which the ADDR designates each memory location to be cleared. This method is fine if the area to be cleared consists of only two or three memory locations, and the clearing operation is required in only one or two different portions of the program. However, if a lengthy table area must be cleared, such as an input buffer (which may store 72 characters or more for a single line of input), this method would be highly impractical. Even for short tables, if they are to be cleared by different routines throughout a program, this method would use up more memory locations than are necessary.

An alternative is to use a subroutine that, when called, will clear as many locations in a table area as defined by the calling program. The routine listed below will fill up to 256 memory locations with zeros. The calling program must set up the index register to the lowest address of the section to be cleared, and accumulator B to the binary count of the number of memory locations in the section. The routine begins at the label CLRMEM. The address of the last location cleared plus one is contained in the index register upon returning.

```
CLRMEM    CLR X        Load memory with 00
          INX          Advance memory pointer
          DECB         Decrement the location counter
          BNE CLRMEM   Counter ≠ 0, continue clearing
          RTS          Return, operation complete
```

TRANSFERRING A SECTION OF MEMORY

Programs of varying applications often have a similar requirement of transferring information in one section of memory to another. For example, an editor program may transfer data from the input buffer to the main text buffer, while a calculator may transfer a multiple precision value from a storage area to a working area in memory. In either case, the programming to perform this function is basically the same. The start address of the section of memory to be transferred and the section of memory to receive the data are set up along with either a count of the number of memory locations to be transferred, or the address of the last location to be transferred.

The first transfer routine uses accumulator B as a binary counter for the number of locations to be transferred. This counter is set up by the calling program, which must also store the lowest address of the section to receive the data in the pair of memory locations referred to here as TEMP2. The index register must be set to the lowest address of the section to be transferred, and will be temporarily stored in the pair of memory locations referred to by the label TEMP1. The index register will indicate the address of the last location plus one transferred by this routine upon returning. The routine begins at the label MOVEIT.

```
MOVEIT    STX TEMP1     Store FROM pointer
          LDAA X        Fetch byte FROM table
          LDX TEMP2     Set up TO pointer
          STAA X        Transfer byte TO table
          INX           Advance TO pointer
          STX TEMP2     Store TO pointer
          LDX TEMP1     Set up FROM pointer
          INX           Advance FROM pointer
          DECB          Decrement byte counter
          BNE MOVEIT    If counter ≠ 0, continue transfer
          RTS           Exit subroutine
```

The next transfer routine checks the contents of the pair of memory locations referred to by the label ADDCHK with the contents of the FROM pointer to determine when the last location has been transferred. This method allows more than the maximum of 256 locations to be transferred, as is the case with the previous transfer routine. When this routine is called, the memory pair ADDCHK must

be set to the last address of the section being transferred, the memory pair TEMP2 must be set to the lowest address of the section to receive the data, and the index register must be set to the lowest address of the section to be transferred. The routine begins at the label MOVEAD and returns with the index register indicating the last location in the FROM table.

```
MOVEAD   STX TEMP2     Store FROM pointer
         LDAA X        Fetch byte FROM table
         LDX TEMP2     Fetch TO pointer
         STAA X        Store byte in TO table
         INX           Advance TO pointer
         STX TEMP2     Store TO pointer
         LDX TEMP1     Fetch FROM pointer
         INX           Advance FROM pointer
         CPX ADDCK     Is end of table reached?
         BNE MOVEAD    No, continue transfer
         RTS           Yes, return to calling program
```

MULTIPLE PRECISION ROUTINES

When dealing with numerical data, it is often necessary to use more than one eight-bit byte to represent a binary number. Since a single byte can only represent a value from 0 to 255, one would be quite limited in the type of calculations that could be performed. This problem is solved by manipulating the data in several bytes as though they were one long register or memory location, N x 8 bits long (N = the number of bytes used to represent the data value). For example, by using two bytes as though they were a single sixteen-bit register, the decimal values from 0 to 65,535 may be represented in binary format. This form of representation is referred to as multiple precision.

In order to perform operations that consider several bytes as one, there must be some link to carry the effects of an operation on one byte over to the next. This link is the carry flag. The carry flag indicates whether an operation on one byte of a multiple precision operation should carry over to the next byte. When the addition of a number to a low order byte of a multiple precision value creates an overflow, the carry flag will be set to a '1' and will be included in the addition of the next higher byte of a multiple precision value. Similarly, when a subtraction requires a borrow for the MSB of a

multiple precision byte, the carry will be set and create a borrow from the LSB of the next higher byte of the multiple precision number.

The subroutines to be described next perform a variety of multiple precision operations on values stored in memory. These operations include incrementing, decrementing, rotating left, rotating right, and complementing a single precision value, and adding, subtracting, and comparing a pair of multiple precision values with each other. For these routines, the multiple precision value(s) is assumed to be stored in consecutive memory locations with the least significant byte in the lowest address.

INCREMENTING A MULTIPLE PRECISION VALUE

There are a number of different reasons why a multiple precision value may have to be incremented. It may be to advance a memory pointer that is stored in memory, or to increment an event counter, or to simply add one to a binary value. For whatever reason, the basic process consists of incrementing the least significant byte and, if it goes to zero as a result, the next byte will be incremented. If this byte also goes to zero, the third byte will be incremented. This process ends when a byte does not go to zero when incremented, or when the most significant byte has been incremented. The first instruction sequence may be used to increment a double precision value by simply loading the index register with the double precision value and using the INX instruction to increment it.

```
LDX MEMADR      Load the double precision value
INX             Increment the double precision value
STX MEMADR      Save the incremented value
```

The next routine increments a multiple precision value stored in memory as indicated by the index register. The number of bytes used for the multiple precision number is set in accumulator B when this subroutine is called. There are two important facts one should be aware of when this subroutine returns. First, the contents of the index register will be pointing to either the last byte that, when incremented, did not go to zero, or to the last location plus one of the multiple precision value. Also, the Z flag will be set to '1' upon returning when the entire value has gone to zero. If any of the bytes do

not go to zero, the Z flag will be '0' when the return is executed. This routine begins at the label INCMEM.

```
INCMEM    INC X        Increment memory contents
          BNE INRET    If result not 0, return
          INX          Advance memory pointer
          DCB          Decrement precision counter
          BNE INCMEM   If counter ≠ 0, continue incrementing
INRET     RTS          Return
```

DECREMENTING A MULTIPLE PRECISION VALUE

The procedure for decrementing a multiple precision value is similar to that for incrementing except for the criteria used to determine when the succeeding byte should be decremented. The next byte is only decremented when the byte being decremented goes from zero to FF, in which case a borrow is required from the next byte. The DEC instruction does not condition the flags to indicate the change from zero to FF, so a different instruction sequence must be used. This sequence uses the SUBA #$01 instruction to decrement a byte because it will cause the C flag to be set to '1' when the zero transition occurs.

The following subroutine decrements a multiple precision value stored in memory. The calling routine must set the index register to the least significant byte of the multiple precision number, and accumulator B to the number of bytes used. The contents of the index register cannot be assumed to point to any one particular memory location upon returning from this routine. This routine begins at the label DCRMEM.

```
DCRMEM    LDAA X       Fetch byte from memory
          SUBA #$01    Decrement the byte
          STAA X       Save byte in memory
          BCC DCRET    If no borrow required, return
          INX          Advance memory pointer
          DECB         Decrement precision counter
          BNE DCRMEM   If counter ≠ 0, continue decrementing
DCRET     RTS          Return
```

ROTATING A MULTIPLE PRECISION VALUE

As the reader familiar with the properties of binary numbers already knows, a binary number can be multiplied by two by simply shifting each bit one position to the left, and loading the LSB with a '0.' And conversely, by shifting each bit of a binary number to the right one bit position, and setting the MSB to '0,' the binary value is divided by two. When rotating a multiple precision number, it is necessary to carry the bit shifted out of a byte over to the next byte. This is done by using the rotate instructions which include the carry flag as part of the accumulator when rotating either left or right. Thus, for a rotate left operation, the MSB shifted out of the lower order byte will be shifted into the LSB of the next byte.

The first routine listed here is labeled the ROTATL subroutine. When this routine is called, the index register must point to the least significant byte of the value to be rotated, and the number of bytes used for the multiple precision value must be in accumulator B. One should note that the initial operation is that of clearing the carry flag. This creates the '0' bit, which must be loaded into the LSB of the multiple precision value. If, for some reason, it is desired to shift a '1' into the LSB, one may set the carry flag before calling this routine, and then enter at the second entry point, labeled ROTL. If the calling program desires to check for a '1' rotated out of the MSB of the value at the completion of the rotate operation, the carry flag will be properly conditioned upon returning to the calling program. Also, the index register will be pointing to the most significant byte.

```
ROTATL   CLC          Clear the carry
  ROTL   ROL X        Rotate byte left
         DECB         Decrement precision counter
         BNE MORRTL   Counter ≠ 0, continue rotate
         RTS          Equals 0, return, operation complete
MORRTL   INX          Advance memory pointer
         BRA ROTL     Continue rotate left
```

The ROTATR subroutine rotates the designated multiple precision value to the right. The index register must indicate the most significant byte of the value when calling this routine, since it works from the most significant byte down to the least significant byte. Accumulator B must be set to the number of bytes in the multiple precision

value. Once again, the carry flag is cleared initially to provide the '0' to be shifted into the MSB of the value. If it is desired to rotate a '1' into the MSB of the most significant byte, the carry flag may be set and this routine may be entered at the second entry point, labeled ROTR.

```
ROTATR   CLC        Clear the carry flag
  ROTR   ROR X      Rotate byte right
         DECB       Decrement precision counter
         BNE MORRTR Counter ≠ 0, continue rotate
         RTS        Equals 0, return, operation complete
MORRTR   DEX        Decrement memory pointer
         BRA ROTR   Continue rotate right operation
```

This final rotate subroutine performs the same operation as the preceeding rotate right subroutine. However, it may be used when the multiple precision value is assumed to be a two's complement number, in which the MSB of the most significant byte is used as the sign bit. This routine makes use of the arithmetic shift right instruction to maintain the initial condition of the sign bit. The setup required for this routine is the same as that for the ROTATR subroutine.

```
ROTARR   ASR X      Rotate right, maintaining sign
 ROTAR   DECB       Decrement precision counter
         BEQ RRET   Counter = 0, return
         INX        Advance memory pointer
         ROR X      Rotate byte right
         BRA ROTAR  Continue rotate operation
  RRET   RTS        Return, operation complete
```

COMPLEMENTING A MULTIPLE PRECISION NUMBER

The complement of a binary value is performed by changing each bit to the opposite condition of its current state. If a bit is a '1,' it is changed to a '0;' if a bit is a '0,' it is changed to a '1.' This type of complement is often referred to as the "one's complement" of a binary number. The one's complement is used, for example, to complement data received from an input device if it is in the opposite state of that required by the program. The complement of the in-

putted data may be derived by a simple COMA or COMB instruction just after reading in the data.

Another form of binary complement is the "two's complement" which may be formed by subtracting the binary number from zero. The two's complement is generally used when a negative value of a binary number is desired. Or, it may be used to form the negative of a subtrahend value that may then be added to the minuend, which effectively subtracts the subtrahend from the minuend. The two's complement of a single byte may be achieved by utilizing the "negate" instruction. For multiple precision values, the following subroutine may be used.

The following routine forms the two's complement of a multiple precision binary number stored in memory. When this routine is called, the index register must indicate the least significant byte of the multiple precision value to be complemented, and the B accumulator must contain the number of bytes defined for the value. This routine uses both the complement and negate instructions. The first byte is negated. Then, if the result of the negate instruction left the byte with a zero value, indicated by a cleared C flag, the next byte will also be negated. If any byte does not go to zero by the execution of the negate instruction, the remaining bytes will be simply complemented. When the routine returns, the index register will be pointing to the most significant byte. This routine begins at the label COMPLM.

```
COMPLM   NEG X        Negate the byte
 COMP1   DECB         Decrement precision counter
         BNE MORCOM   Counter ≠ 0, continue
         RTS          Counter = 0, return
MORCOM   INX          Advance memory pointer
         BCC COMPLM   Last byte = 0, negate next
         COM X        Not 0, complement byte
         BRA COMP1    Continue operation
```

MULTIPLE PRECISION ADDITION AND SUBTRACTION

Addition and subtraction are common functions often required when dealing with multiple precision values that represent numeric data. Both operations work from the least significant byte up to the

most significant byte, using the carry flag as the link between bytes. When the addition of two bytes results in an overflow from the MSB, the carry flag is set and is included in the addition of the LSB's of the next pair of bytes. Similarly, if the subtraction of a pair of bytes results in a borrow required from the next byte of the minuend, the carry flag is set, which causes a borrow from the LSB of the next byte of the subtraction.

The addition routine is labeled ADDER. This routine adds the multiple precision value indicated by the pointer in TEMP2 to the value indicated by the index register, with the result of the addition stored in place of the value indicated by the index register. The index register and TEMP2 must be set to the least significant bytes of the multiple precision numbers. Accumulator B must be set to the binary count of the number of bytes in the multiple precision values. The carry flag will indicate whether an overflow from the MSB of the most significant byte has occurred upon returning from this routine. The calling routine may have to check this flag since an overflow would usually indicate an error condition. At the completion of this routine, the index register will be pointing to the most significant byte plus one of the result.

```
ADDER   CLC             Clear carry flag
        LDAA X          Fetch byte from first value
        STX TEMP1       Save pointer
        LDX TEMP2       Fetch pointer to second value
        ADCA X          Add byte from second value
        INX             Advance pointer to second value
        STX TEMP2       Save pointer to second value
        LDX TEMP1       Fetch pointer to first value
        STAA X          Store sum of bytes
        INX             Advance pointer to next byte
        DECB            Decrement precision counter
        BNE ADDER+$1    ≠0, continue addition
        RTS             = 0, return to calling program
```

The subtraction routine, labeled SUBBER, subtracts two multiple precision values stored in memory. TEMP1 must indicate the least significant byte of the minuend; the index register must indicate the least significant byte of the subtrahend; and accumulator B must contain the binary count of the number of bytes in each multiple precision value when this routine is called. The result of the subtrac-

tion is stored in place of the subtrahend, and the index register will indicate the most significant byte plus one of the minuend upon returning to the calling program. The carry flag will indicate whether a borrow was required by the subtraction of the MSB of the most significant byte. This routine begins at the label SUBBER.

```
SUBBER  CLC             Clear carry flag
        LDAA X          Fetch byte from minuend
        STX TEMP1       Save pointer to minuend
        LDX TEMP2       Fetch pointer to subtrahend
        SBCA X          Subtract byte of subtrahend
        STAA X          Store result in subtrahend
        INX             Advance subtrahend pointer
        STX TEMP2       Save pointer to subtrahend
        LDX TEMP1       Fetch pointer to minuend
        INX             Advance minuend pointer
        DECB            Decrement precision counter
        BNE SUBBER+$1   ≠0, continue subtraction
        RTS             =0, return with result in subtrahend
```

COMPARING TWO MULTIPLE PRECISION VALUES

It is often desired to determine whether one number is larger or smaller in magnitude than another. This fact may change the manner in which a program is to deal with two numbers. For example, when subtracting two numbers, it is usually necessary to subtract the larger from the smaller and, if indicated, change the sign of the result. The following routine may be used to compare two multiple precision numbers stored in memory.

This same routine may be used to compare alphabetic information, such as one name against another, to test for duplication or, if the character set is well ordered (as is the case with the ASCII code), to place the names in alphabetical order.

The compare routine presented next compares the multiple precision value indicated by the pointer in TEMP2 against the value indicated by the pointer in TEMP1. These pointers must be initially set to the most significant byte of the respective values to be compared. The B accumulator must be set to the binary count of the number of bytes to be compared. Upon returning, the carry and zero flags will be set to indicate the outcome of the comparison. The

calling program must check these flags by using the BHI, BEQ or BLS conditional branch instructions to branch to the desired routine as a result of the comparison. The index register will indicate either the least significant byte of the value being compared, if the two values are equal, or the byte at which the comparison failed. This routine begins at the label CPRMEM.

(The two instructions marked by the @@ may be changed to INX for comparing alphabetic data, if the data is arranged with the first character in the lowest address for each of the character strings being compared.)

```
CPRCON   DEX          @@  Advance compare TO pointer
         STX TEMP2         Save compare TO pointer
CPRMEM   LDX TEMP1         Fetch compare pointer
         LDAA X            Fetch compare data
         DEX          @@   Advance compare pointer
         STX TEMP1         Store compare pointer
         LDX TEMP2         Fetch compare TO pointer
         CMPA X            Compare A to indexed data
         BNE CPRET         If not equal, return
         DECB              Decrement byte counter
         BNE CPRCON        Counter ≠ 0, continue comparing
CPRET    RTS               Return with flags conditioned
```

The following listing illustrates a possible instruction sequence for calling the CPRMEM routine, and then checking the results of the compare operation for one of the three possible conditions.

```
...              Instructions leading up to the compare
LDX TBL1         Set up pntr to value to be compared
STX TEMP1        Store this pointer in TEMP1
LDX TBL2         Set pntr to value to be cmprd against
STX TEMP2        Store this pointer in TEMP2
LDAB #$XX        Set precision counter to no. of bytes
BSR CPRMEM       Compare TEMP1 against TEMP2
BHI GRTRTN       TBL1 grtr thn TBL2, jmp to GRTRTN
BEQ EQUAL        TBL1 = TBL2, jmp to EQUAL
...              TBL1 less than TBL2, begin processing
                 For less than
```

CHECKING FOR VALUE WITHIN LIMITS

Another type of comparison often required is to check whether the value of a byte of data falls within expected limits. One frequent application is in checking the code received from an input device. For example, a calculator program may check each character input for a legal digit code when it expects to be receiving only digital information, or a control program may check inputs from a sensing device to determine whether the parameter it is checking is within allowable limits. When the data being checked is to fall within sequential limits (limits defined by an upper and lower bound), the following type of routine may be used.

The routine compares a byte of data against the lower limit, and then the upper limit plus one, of the boundaries in which the data byte must fall. The reason for checking the upper limit plus one is to allow the condition of the carry flag, upon returning, to indicate whether the byte falls within the designated limits. When the routine returns to the calling program with the carry set, the byte is not within the limits. When this routine is called, the data byte to be checked must be in the A accumulator.

The routine listed below checks for the ASCII code for the digits 0 through 9, namely 260 to 271. If one desires to use this program to check for the ASCII code for the alphabetic characters A through Z, for example, the immediate portion of the compare instructions would simply be changed to C1 (ASCII A) and DB (1 greater than ASCII Z). This routine begins at the label LMTCHK.

```
LMTCHK    CMPA  #$B0     Is byte less than ASCII 0?
          BSC LMTRET     Yes, not in limits, return with C =1
          CMPA  #$BA     Is data byte greater than ASCII 9?
          BSC CRCLR      If not, reset C to 0 before returning
          SEC            If so, return with C = 1
LMTRET    RTS            Return to calling program
 CRCLR    CLC            Within limits, return with C = 0
          RTS            Return
```

PROGRAMMED TIME DELAYS

The computer is designed to execute a program stored in its memory as rapidly as it possibly can. It does not hesitate between instruc-

tions, as a human would, to contemplate the next operation it should perform. However, there are certain types of programs that require a hesitation, or delay, between one operation and the next. One type of program might be a display program that outputs a frame of characters or pattern to a video device, and then must wait a specific amount of time before outputting the next frame or pattern. Or, a delay may be required after outputting a control command, which turns on a motor driven device, to allow the motor to get up enough speed before a data transfer may be initiated with the device. A programmed delay may also be required between outputting each bit of a serial data pattern to allow the program to control the data transmission rate. By inserting program time delay sequences, one may affect these REAL TIME program applications.

As pointed out at the end of Chapter one, each instruction requires a specific number of cycles and, therefore, a specific amount of time to execute. By knowing the exact time for each instruction, a delay may be created by programming a group of instructions whose total execution time is as close to the desired delay as possible. (For the 6800 with a clock frequency of one megahertz, one should be able to program a delay to within two microseconds of the desired time.) Also pointed out is the fact that, depending on the type of memory used in one's system, the actual timing of the instructions may vary from those presented in Appendix A. Before getting into the time delay programming, it is necessary to understand the differences between various types of memories so that one will be able to discern the actual timing for one's own system.

(The following description is presented in general terms to enlighten the reader with the basic theory of memory acquisition. It is not intended to educate the reader in the specific details of hardware requirements for accessing memory. One should refer to the manual supplied by the hardware manufacturer for one's particular computer for details on memory accessing.)

When a computer must access a memory location, to read an instruction or data from it, or to write data into it, the address of the memory location is first placed on the memory address buss. Then, after the memory has been given time to select the memory location accessed, the contents of the location may be read from the data buss by the CPU, or the contents of the data buss may be written into the memory location. The length of time required to access a

memory location for reading or writing is referred to as the speed of the memory. Depending on the speed of the memory, the delay required between sending the address and accessing the memory location may vary. If a normal delay in the CPU instruction time is sufficient for the memory to react to the address selection, the data may be read or written during the second portion of the cycle. If, however, the memory cannot react fast enough, the second portion of the cycle must be drawn out to allow enough time to pass before reading from or writing to the memory location.

The 6800 extends the second portion of the cycle by stretching out the second phase of the CPU clock to allow time for slower memory to access the proper location. The amount of time added to this second phase may vary from one manufacturer to another, or possibly from one computer to another, depending on the speed of the memory used. The reader should refer to the description on memory read and write operations that should be included in the hardware manual supplied by the manufacturer, to determine the exact amount of time used for the second phase of the CPU clock. Knowing the timing used by one's computer will allow one to program exact time delays on a system that uses ROM or static RAM memory.

The use of ROM or static RAM memory allows one to determine the exact timing required for each instruction, although it may be different from the times presented in Appendix A. Each instruction requires a given number of CPU cycles for its execution. Multiplying the number of CPU cycles times the length of time for each cycle will give one the exact time required to execute an instruction. If the second phase of the CPU clock is extended to allow for slow memory, this additional time must be added to each cycle executed for an instruction.

The use of dynamic RAM memory in a system makes it difficult to calculate the instruction timing accurately. The reason for this is that the dynamic RAM memory requires a refresh cycle at least once every one or two milliseconds. (This time may vary for different types of dynamic memory.) A refresh cycle means that within the allotted time, each memory address must be accessed with a read cycle in order for the memory to maintain its current contents. This refresh process may interrupt the timing of the CPU instructions, since the refresh circuitry may be accessing a memory location at the

same time the CPU may require a memory access. In this case, the CPU would have to wait for the refresh read to complete, thereby extending the time required for the instruction to execute. It is, therefore, only possible to calculate a minimum time delay for a given instruction sequence, and not the maximum when using dynamic RAM memory.

With a knowledge of the timing necessary for the instructions, one may begin to program time delays of specific duration. In programming a delay, one should strive to use as few instructions as possible, and care must be taken to insure that the instructions used in the delay do not interfere with the operation of the main program. Several forms of delay routines will now be presented. Unless stated otherwise, the times given are those listed in Appendix A, which assumes no delay has been added to the second portion of the clock cycle.

For very short delays, in the order of two to twenty microseconds, several instructions that fall in the direct sequence of the program may be used. Suppose a delay of six microseconds is required at a certain point in a program. A "no operation" instruction requires two microseconds to execute, so the desired six microsecond delay may be derived by using three NOP instructions at the point in the program requiring the delay.

Another method used to create short delays is to insert a jump to subroutine that jumps to a location that contains a return instruction. This sequence would delay nine microseconds for the jump to subroutine instruction plus five microseconds for the return, for a total of fourteen microseconds. To conserve memory, the return instruction may be part of an existing routine. It need not be set up as a return specifically for this delay.

For longer delays, the method of inserting the delay instructions in sequence with the main program would begin to waste a great deal of memory. An alternative is to use a subroutine that will form a timing loop to delay the desired amount of time. The following routine allows control of the delay time by selection of an initial value for the B accumulator. The delay is created by forming a program loop that decrements the B accumulator until it reaches zero, and then returns. The larger the initial value of the B accumulator, the longer the delay; the exception to this is that the initial value of zero will create the longest delay.

```
DELAY   DECB          Decrement delay cntr (2 u'sec)
        BNE DELAY     If cntr ≠ 0, loop back (4 u'sec)
        RTS           Counter = 0, return (5 u'sec)
```

The amount of time used by this routine is calculated by adding up the time required for each instruction every time it is executed. The execution time for each instruction is given in parenthesis after each comment. The following formula may be used to calculate the delay time for a given value of accumulator B. If accumulator B is initially zero, the value of 256 must be substituted in this equation.

```
                  LDAB  BSR  DECB  BNE                RTS
DELAY TIME = 2  +  8  + (2  +  4) * (ACC B) + 5
```

One fact presented in this formula is that the time required for the instruction LDAB and BSR, which set up the B accumulator to the required constant and then called the DELAY subroutine, must be included in the calculation of the delay time. The time given for the LDAB instruction assumes the immediate addressing mode. Also, the time required for setting up the B accumulator may be excluded if it is set up previous to the time the delay is to begin.

The time delay that can be created by this program loop runs from a minimum of 21 microseconds for the B accumulator equal to one, to a maximum of 1551 microseconds, for the B accumulator equal to zero in increments of six. This incremental factor is controlled by the loop DECB, BSR. Should it be desired to expand the increment, and thereby extend the maximum delay possible, additional instructions may be added to this loop. For example, if the incremental factor is desired to be eight microseconds rather than six, a NOP instruction can be inserted between the DECB and BNE instruction, which will add two microseconds to the loop without altering the basic operation of the routine.

When the actual delay required by a program does not equal one of the incremental times generated by the delay loop, the delay may be adjusted by setting the B accumulator to the closest incremental time without exceeding the time desired, and then adding one or two instructions to the calling sequence to bring the total delay to within one or two microseconds. For example, suppose a delay of 427

microseconds is needed. Selecting a value of 68 for the B accumulator will provide a delay of 423 microseconds. The additional four microseconds can be made up by adding two NOP instructions to the calling routine before the BSR DELAY instruction. These additional instructions will add the necessary four microseconds to the total delay.

Substantually longer timing loops can be derived by nesting delay loops. Using both accumulators, one can set up a delay loop within a delay loop. Then, when one loop goes through a complete cycle, the second loop will be decremented once, thereby multiplying the time required for the inside loop by the initial value (minus one) of the accumulator in the outside loop. The following routine, which includes the calling sequence, illustrates this method of nesting delay loops. The B accumulator is used as the counter for the inside loop and the A accumulator is used for the outside loop. Here again the greater the initial value of the accumulator, the longer the delay, with the exception of the value zero, which creates the longest delay. The exact timing for this routine will be discussed following the listing.

```
            ...
            LDAB  #$XX      Set initial inside loop (2 u'sec)
            LDAA  #$YY      Set initial outside loop (2 u'sec)
            BSR DLYLOP      Call delay loop routine (8 u'sec)
            ...

DLYLOP      DECA            Decrement outside loop (2 u'sec)
            BNE DLYLP1      If ≠0, branch to inside loop (4 u'sec)
            RTS             If = 0, return, delay over (5 u'sec)
DLYLP1      DECB            Decrement inside loop (2 u'sec)
            BEQ DLYLOP      If = 0, branch to outside loop (4 u'sec)
            BRA DLYLP1      If ≠0, continue inside loop (4 u'sec)
```

Calculation of the amount of time this routine will require for execution can be made from the formula given next. This formula is shown in two forms. One indicates the instruction sequence that is executed, and the second provides a condensed version for use in making the actual calculation.

```
                LDAB LDAA   BSR
DELAY TIME =     2  +  2  +  8  +

                DECA BNE                DECB BEQ  BRA DECB BEQ
                [ 2  +  4  +((INIT B)-1)*(2  +  4  +  4)+  2  +  4 ] +

                          DECA BNE      DECB BEQ  BRA DECB BEQ
                [((INIT A)-2)*( 2  +  4 +(255*( 2  +  4  +  4)) + 2  +  4]+

               DECA BNE RTS
                2  +  4  +  5
```

DELAY TIME = (((INIT A)-2)*2556)+(((INIT B)-1)*10)+41

The first formula has two sections enclosed in brackets. The first bracketed section indicates the time for the first pass through the inside loop, and the second bracketed section indicates the time for all successive passes. The reason for the separation of these times is that on the first pass through the inside loop, the initial value of the B accumulator will be initialized by the calling program. After the first pass, the B accumulator will always be zero when the inside loop is entered. This formula is only valid for initial values of the A accumulator from 2 to 256. (In actual operation of the subroutine, the accumulators are initialized to 0 when 256 is used in the formula.) If the A accumulator is initially set to 1, the execution time is simply the sum of the times not enclosed in the brackets, which is 23 microseconds. For values of the A accumulator from 2 to 256, the time delay can be set within the limits of 41 to 651,815 microseconds in intervals of 10 microseconds. If finer selection is required, the technique discussed previously of inserting instructions in the calling sequence may be used.

RANDOM NUMBER GENERATORS

The purpose of a random number generator is to provide a non-repeating series of random numbers. These random numbers may be applied to a number of different programs. For instance, when a game (such as dice or blackjack) is programmed, the program must provide a random assortment of numbers for the roll of the dice or a draw of a card. This is accomplished by using some form of a random number generator routine. Another application for random number generators is in creating random patterns for testing devices, such as

a computer's memory, which may be sensitive to various random patterns.

Two methods of programming random number generators will now be presented. The first is a very simple method that may be used when the numbers are required only in response to an input from the program operator. The second method uses a routine that will produce a new random number each time it is called.

When a program requires a random number only in response to an input received from the operator, the random number may be derived by constantly incrementing an accumulator or memory location until the input is received. This may be accomplished by forming a program loop that increments the register and then checks the status of the input device for an input from the operator. If the status indicates there is no input received, the routine will loop back to increment the accumulator or memory location again. Since the program loop is so short, probably in the range of 30 to 50 microseconds, it would be impossible for a human to select the precise time to input a character, thereby stopping the loop when a specific value is present. So, for programs that require random numbers in response to operator inputs, the following routine may be used to generate the random numbers. The CHKINP called in this program is assumed to check the status of the input device and return with the sign flag set to '1' if a character has been entered on the keyboard, or set to '0' if a character has not been entered. When the character has been received, the value in the B accumulator may be used as the random number for the program.

```
RNDMLP  INCB        Increment random number
        BSR CHKINP  Check for character entered
        BPL RNDMLP  Not entered, increment again
                    When entered, use accumulator B for
        ...         Random number
```

When a program requires random numbers at various times throughout its operation, not necessarily in response to an input, the following routine may be used. This routine generates a pseudo-random data pattern of eight bit bytes. This random number generator is not a true generator because, depending on its initial values, it will create a repetitive pattern every 1000 to 4000 num-

bers. However, the program that uses it can make the data more random by using a trick that will be described shortly.

This random number generator uses two consecutive memory locations (other than those required by the routine itself) to save the random number and an incrementing addend. Each time the routine is called, the random number created last is used in generating the next random number. It is operated on by the series of instructions in this routine, and then the addend is added to it to create the new random number. At the same time, the addend will be incremented either once or twice, depending on the result of the addition of the random number to the addend, as indicated in the listing. The new random number will be saved in memory and returned to the calling program in the A accumulator.

The trick referred to previously for increasing the randomness of the numbers generated is to have the calling program alter the contents of the addend at a point in the program that is occasionally executed. For example, if the program enters a subroutine once for every ten or fifteen times it calls the random number generator, an instruction sequence should be added to the subroutine to alter the addend. It may be altered by incrementing once, or adding five, or resetting the addend to zero. No matter what method is used to change the value of the addend, the result will be that of altering the data pattern generated, since the normal sequence of the addend will be disrupted.

The listing for the random number generator is given next. It begins at the label RANDOM. The contents of the index register will indicate the memory location containing the random number upon returning to the calling program. The instruction sequence that follows this routine may be used by the calling program to alter the addend by adding five to it.

```
RANDOM   LDX RANNUM     Set pointer to random number
         LDAA X         Fetch random number
         ROLA           Perform a series of
         EORA X         Operations on it to
         RORA           Create a new random number
         INC $1,X       Increment the addend
         ADDA $1,X      Add the addend
         BVC SKIP       If V = 0, increment addend once
         INC $1,X       Otherwise, increment it twice
```

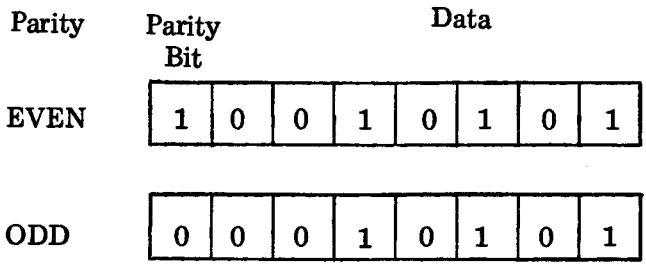
```
SKIP    STAA X          Store new random number
        RTS             Return to calling program

        ...             Sequence to add 5 to addend
        LDAA ADDEND     Fetch random number addend
        ADDA  $5        Add 5 to addend
        STAA ADDEND     Save new addend
        ...             Continue processing
```

CHECKING PARITY

The transmission of data from one device to another is at some time subject to some form of transmission error. This error may be due to an intermittent problem with the equipment transmitting or receiving the data, or random noise that may occur, for example, when the data is transmitted over the telephone lines. No matter what the reason for this error, it is often desirable to set up some method of testing for such an error.

One method often used is referred to as checking for parity. The data is broken down to small groups, such as eight bits at a time, in which seven bits contain the data to be transmitted, and the eighth bit is used as the parity bit. The condition of the eighth bit is determined by the number of bits among the other seven that are set to one, and by the selection of either odd or even parity. Odd parity simply means that there must be an odd number of bits set to one for the parity to be correct, and even parity means that there must be an even number of bits set to one. Thus, the condition of the parity bit is set or reset to make the total number of bits set to one in the given group odd or even, depending on the odd or even parity requirements. For example, if three out of seven data bits are set to one, and the group is to have even parity, the parity bit must also be set, resulting in four bits set to one. If the group is to have odd parity, the parity bit would be reset to zero so that the total number of bits set to one would remain an odd number. These examples are illustrated on the following page.

The following routine checks the parity of an eight bit group. It may be used for either determining the condition of the parity bit for outputting the data, or testing the parity of data received. When calling this routine, the data to be checked must be in the A accumulator. The parity is checked by rotating each bit into the carry and incrementing a parity count for each bit found to be one. The parity



count is stored in a separate memory location referred to by the label PTYCNT. After each bit is tested, the least significant bit of the parity count is loaded into the C flag. When the routine returns, the A accumulator will maintain its initial contents, and the C flag will be set to one if the data has odd parity, or reset to zero if the data has even parity. If this routine is used to determine the setting of the parity bit for outputting the data, the parity bit should be zero when calling this routine, and then conditioned to create the desired parity by checking the C flag upon returning. The instruction sequence that follows this listing illustrates a method of setting up data to be transmitted with even parity. The MSB is assumed to be the parity bit. For checking the parity of data received, the data is simply loaded into the A accumulator, this routine is called, and, upon returning, the C flag is tested.

```
PARITY  LDAB #$08       Set bit counter
        CLR PTYCNT      Clear parity counter
LOOP    ROLA            Rotate bit into carry
        BCC ZEROBT      Bit = 0, don't increment  parity counter
        INC PTYCNT      Bit = 1, add 1 to parity counter
ZEROBT  DECB            Decrement bit counter
        BNE LOOP        ≠0, continue parity check
        ROLA            Rotate once more to restore data
        ROR PTYCNT      Rotate LSB of parity cntr into carry
        RTS             Ret, C=1 odd parity, C=0 even parity


        ...
        BSR PARITY      Check parity of data to be transmitted
        BCC EVEN        Even parity, output data as is
        ORAA #$80       Odd parity, set parity bit to make even
EVEN    ...             Proceed to output data
```

CONVERSION ROUTINES

The real power provided by a computer is exemplified by its capability to operate with unlimited variations of character codes by simply changing a program. It can accept information in one form, convert it to another for processing, and then output it in the same format as initially received, or in an entirely different format. One may be aware of various other devices that perform such conversions, however, the input and output codes are most likely fixed, allowing no variation. A computer may be programmed to utilize a variety of codes for input, processing, and output.

The need for code conversion and the type of conversion required is governed by two factors. The code used by the peripheral devices for transmitting and receiving data is one factor. If the input device transmits one code, and the output device must receive a different code, conversion from one to the other must be made at some point in the program. The other factor is the format required by the program to process the data. If this format is the same code as received from the input device, no conversion is required. Otherwise, the appropriate conversion must be made before the program can proceed with its processing.

The codes used by different I/O devices to transmit and receive data can vary from a standard code, used to represent a specific character set, to a code that has been designed into a special purpose device. Several of the standard codes used to represent alphanumeric information are ASCII, BAUDOT, EBCDIC, and HOLLERITH. ASCII and BAUDOT are commonly used on keyboard and printer or display devices, such as CRT terminals and teletypewriter machines. EBCDIC is generally used for mass storage devices, such as magnetic tape units, and HOLLERITH is generally associated with card reader/punch devices. The special purpose codes may be derived, for example, when interfacing a calculator-type keyboard to enter both numeric data and the functions that are to operate on the data.

The code used by a program to process the information input may be the same code as received, or it may require some conversion to a format convenient for the computer to operate with. When a pro-

gram deals with the manipulation of text, such as an editor program, the character code received by the program is often used for storing the text information. As each character is input, the code is stored as the representation to be used by the program for that character. For programs that deal with numeric data, for arithmetic operations or designating digital information, conversion from the character code received to the binary or decimal equivalent may be required. A calculator program might receive the data as ASCII encoded decimal digits that must be converted to the binary equivalent for processing, and then back to ASCII digits to output the answer. A monitor program may require the conversion of the coded octal or hexadecimal input to the binary equivalent for defining memory addresses and their contents.

As mentioned previously, there are a number of standard codes used to transfer data from a peripheral device to a computer and vice versa. For the following discussion on conversion from one code to another, the ASCII and BAUDOT codes will be used. Their contrasting formats aid in describing various methods of code conversion. Therefore, to preface the conversion routines, a brief discussion of each code is presented.

The ASCII code is a seven bit code that has codes representing the entire alphanumeric character set plus punctuation marks and a number of non-printing control characters. An eighth bit is often added to this code. This bit can be used to provide parity, for error checking, or it can be set to a constant '1' or '0' condition for all characters. The ASCII codes for the printing characters and several of the control characters are presented in Appendix D, in both octal and hexadecimal notation. The code presented in Appendix D, and used throughout this book whenever ASCII is discussed, assumes the eighth bit is always set to '1.'

As the reader may notice by examining Appendix D, the ASCII code is a well-ordered code. The letters of the alphabet are represented in sequential order from 301 (octal) for A to 332 (octal) for Z. The numbers are similarily ordered from 260 (octal) to 271 (octal) for the numbers 0 through 9, respectively. This coding for the numbers allows ease of conversion from ASCII to binary-coded-decimal by simply dropping the 4 most significant bits of the ASCII

code. The convenience of the ASCII code in this format is in sharp contrast to the BAUDOT code, discussed next.

The BAUDOT code is a five bit code used to represent the alphanumeric character set plus several punctuation marks and control characters. Appendix E contains the octal representation of the BAUDOT code. The code presented assumes the three most significant bits are all '0.' The reader unfamiliar with BAUDOT may question how 5 bits can be used to represent over 32 characters. The answer is quite simple. Each of the letters of the alphabet shares its code with a numeral or punctuation mark. Two separate control characters are used to determine which of the two possible characters is being transmitted, one indicating the letters and the other indicating the figures (numerals or punctuation marks). The proper mode (letters or figures) must be set by outputting the corresponding control character before the output of one or more of the characters of that mode. For example, if a sentence consisting entirely of letters were to be typed on a BAUDOT keyboard, the letters control character would be entered first, followed by the letters that make up the words of the sentence, and then to end the sentence, the figures control character would be entered followed by the period, which shares its code with the letter M. It should be noted that the codes for the space, carriage return, line feed, and null characters are common to both modes.

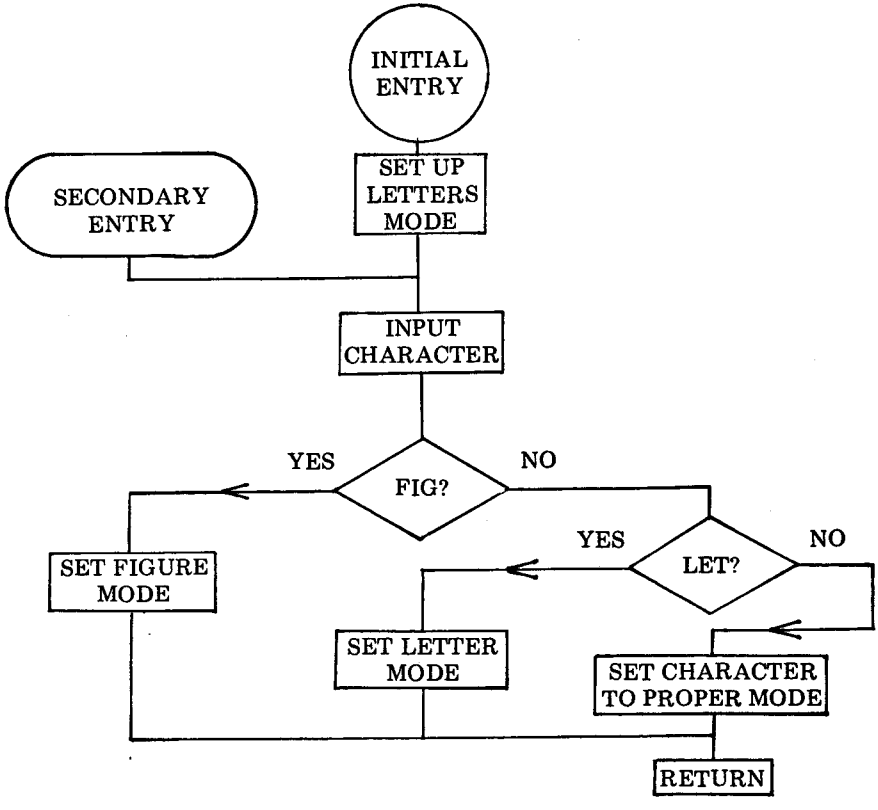
Examination of the BAUDOT code in Appendix E reveals the obvious scrambled pattern of character codes. There is no set pattern that would lend itself to ease of recognition of the BAUDOT letters, as there is with the ASCII code. And, conversion of the BAUDOT code for the numerals to the equivalent BCD values is certainly not as trivial as conversion of the ASCII digits described previously.

Programs that operate with the BAUDOT code to represent the character set, or perform conversion to or from another code, must have some means of differentiating between the two characters a BAUDOT code may represent. This can be accomplished by defining one of the three most significant bits as a mode designator. That is, one of these three bits would be set to '0' for the letter mode, and to '1' for the figure mode. For this discussion, bit five will be so designated. The following pair of routines may be used to encode and decode the BAUDOT characters according to this method for separating the letters from the figures.

The first routine is used to encode the BAUDOT characters as they are input. There are two entry points for this routine. The first, labeled BAUDIN, is used when the input of characters is to be initialized. The initialization is done by outputting a letters control character to the printer device before calling the input routine to receive a character from the keyboard, and setting up a memory location to indicate the current mode of the printer device. This memory location, labeled CNTRL, is conditioned by the receipt of the letters or figures control characters, and is used to encode the characters as they are received. The other entry point at label INBAUD is used after the initialization has been completed. This entry point assumes that CNTRL is properly conditioned. The routine returns to the calling program with the character contained in the A accumulator. The listing and flow chart for this routine are now presented.

```
BAUDIN   LDAA  #$1F     Load letters code into accumulator
         JSR OUTPUT     Call routine to send BAUDOT char
         BSR LETCOD     Initialize CNTRL to letters code
INBAUD   JSR INPUT      Now accept BAUDOT char from kybd
         CMPA  #$1B     See if figures code
         BEQ FIGCOD     Yes, set CNTRL to 20
         CMPA  #$1F     See if letters code
         BEQ LETCOD     Yes, set CNTRL to 00
         ADDA CNTRL     Add in condition of 6th bit
         RTS            Return to process data

FIGCOD   LDAB  #$20     Set 6th bit of CNTRL
         STAB CNTRL     By loading it with 20
         RTS            Return to process data

LETCOD   CLR CNTRL      Reset 6th bit of CNTRL
         RTS            Return to process data
```

Two subroutines are called out in this listing to perform the input and output operations with the BAUDOT devices. The INPUT routine inputs a character from the BAUDOT keyboard, and returns to this routine with that character in the A accumulator. The OUTPUT routine must transmit the character contained in the A accumulator to the BAUDOT output device. The reader may refer to the chapter on I/O processing for methods of implementing these INPUT and OUTPUT routines.

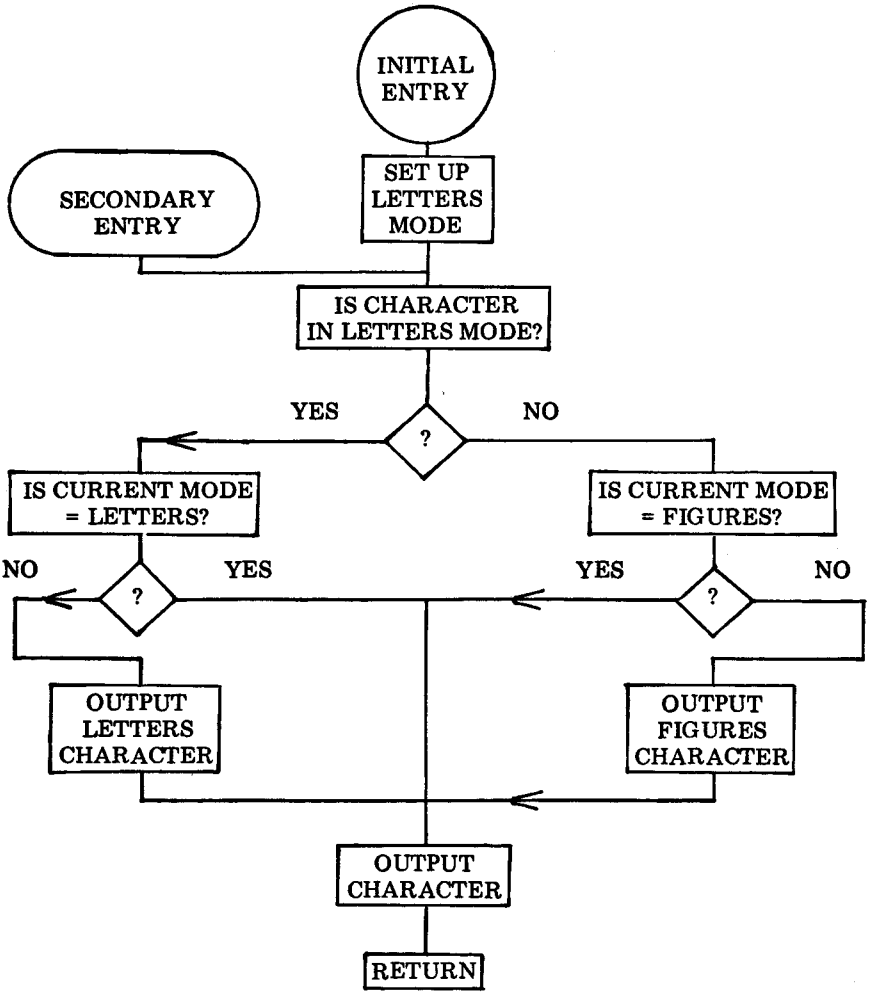


The following BAUDOT output routine may be used to decode BAUDOT characters before outputting. This routine also has two entry points. The BAUDOT entry point is called when the initial character of the string of characters is to be outputted. This entry point sets up the output device, and the CNTRL memory location to the letters mode before outputting the character. After the first character, the subsequent characters are output by using entry point OTBAUD. OTBAUD first checks the character to be output for a change from the current mode, and, if different, the proper mode control character will be output before the character. The character to be output must be stored in the B accumulator before calling either of these entry points. The OUTPUT routine must function in the same manner as previously described. The flow chart for this routine follows the listing.

```
BAUDOT   LDAA  #$1F    Load letters code into 'A'
         JSR OUTPUT    Call routine to send BAUDOT code
         CLR CNTRL     Reset CNTRL to letters mode
OTBAUD   BITB  #$20    Is 6th bit = to 1?
         BEQ LTCHAR    If not, character is a letter
         TST CNTRL     If figures, see if last output was figures
         BEQ LASLET    No, must output figure code
OUTCOD   TBA           Put character in 'A' accumulator
         JSR OUTPUT    Send the BAUDOT character
         RTS           Return to calling routine

LASLET   LDAA  #$20    Last was letter, set CNTRL
         STAA CNTRL    To indicate figure code
         LDAA  #$1B    Output figure code
LASFIG   JSR OUTPUT    Send control character
         JMP OUTCOD    Now send character

LTCHAR   TST CNTRL     See if last was letter
         BEQ OUTCOD    Yes, output character
         CLR CNTRL     Reset CNTRL to letter mode
         LDAA  #$1F    Set up to output letter mode
         BRA LASFIG    By using routine above
```

Using the ASCII and BAUDOT codes as the sample codes, two methods of code conversion will now be presented. The first method uses a look-up table. The table consists of both the ASCII and BAUDOT codes for each character of the character sets. The entries in the table are arranged in pairs. The first entry of a pair contains the ASCII code for the character, and the second entry contains the BAUDOT code for the same character. In cases where there is no equivalent BAUDOT code for a character, an appropriate substitute may be inserted (for example, the BAUDOT code for the left and right parenthesis, (and), may be substituted as the equivalent code for the ASCII left and right brackets, [and]), or the BAUDOT null character may be used when no suitable substitute is available. Sample entries for this table are presented next.

The conversion program that uses this table begins at one end of the table and compares the character code to be converted against the entries in the table of the same character set. For conversion from ASCII to BAUDOT, the ASCII code to be converted is compared to the ASCII entries in the table. When a match is found, the BAUDOT entry of the pair is returned as the BAUDOT equivalent. A similar process is used to convert BAUDOT to ASCII. The flow chart on the following page indicates the logic used for conversion in either direction.

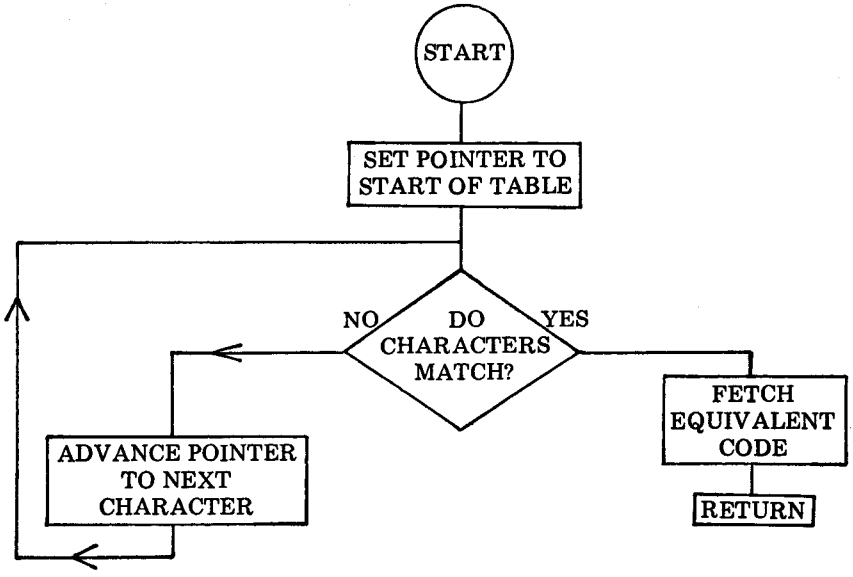


ADDRESS		HEXA CODE	
0700	ASBDTB,	C1	ASCII A
0701		03	BAUDOT A
0702		C2	ASCII B
0703		19	BAUDOT B
.		.	
.		.	
.		.	
073E		A0	ASCII SPACE

```
073F            04      BAUDOT SPACE
0740            A1      ASCII !
0741            2D      BAUDOT !
0742            A2      ASCII "
0743            31      BAUDOT "
0744            A3      ASCII #
0745            34      BAUDOT #
  .             .
  .             .
075E            AF      ASCII /
075F            3D      BAUDOT /
0760            DB      ASCII [
0761            2F      BAUDOT ( (substitute)
0762            DD      ASCII ]
0763            37      BAUDOT ) (substitute)
0764            A8      ASCII (
0765            2F      BAUDOT (
0766            A9      ASCII )
0767            32      BAUDOT )
  .             .
  .             .
077C            BF      ASCII ?
077D            39      BAUDOT ?
077E            C0      ASCII @
077F            00      BAUDOT NULL (substitute)
0780            FF      ASCII RUBOUT
0781  BDASTB,    00      BAUDOT NULL
```

Since the conversion begins at one end of the table, when substitute characters are used, the true code conversion pair must be located such that it will be found before the substitute codes are encountered. For example, in the table given previously, the ASCII and BAUDOT pairs for left and right parenthesis must be placed in the table so that conversion from BAUDOT to ASCII will result in the ASCII code for the left and right parenthesis being returned, and not the ASCII code for the left and right brackets. The correct location for these codes is illustrated in the table sample.

The listings for the conversion routines from ASCII to BAUDOT, and vice versa, using the look-up table, are given next. While examining these routines, it should be noted that they both assume that the code to be converted is contained in the accumulator when the routine is called. It may also be noted that the ASCII to BAUDOT routine searches the table from the low address end, while the BAUDOT to ASCII routine begins at the high address end of the table.




```
ASBAUD  LDX #ASBDTB   Set pointer to low address end of table
 FASCII CMPA X        Compare ASCII code to 'A'
        BEQ FNDBDO    If match, do conversion
        INX           No match, advance pointer
        INX           To next ASCII code
        BRA FASCII ++ And continue looking
FNDBDO  LDAA #$1,X    Fetch BAUDOT equivalent from table
        RTS           Return with code in 'A'

BAUDAS  LDX #BDASTB   Set pointer to high address end of tbl
FBAUDO  CMPA X        Compare BAUDOT code to 'A'
        BEQ FNDASC    If match, do conversion
        DEX           No match, decrement pointer to
        DEX           Next BAUDOT code
        BRA FBAUDO ++ And continue looking
FNDASC  DEX           Back up pointer to ASCII code
        LDAA X        Fetch ASCII equivalent from table
        RTS           Return with code in 'A'
```

Both routines just presented assume that the code to be converted is a valid code that is included in the look-up table. If for some reason the code in the accumulator is not a valid code, an end of table test should be added to the conversion routines. The following routine may be inserted into either routine to test for the end of the table by replacing the instructions marked by the ++ with this instruction sequence. The immediate portion of the CPX instruction must be set to the address that will indicate the end of the table. For the ASBAUD routine, this would be the address of BDASTB +1, and for the BAUDAS routine the address would be ASBDTB -1. If the end of the table is reached, the A accumulator should be loaded with an error code to indicate to the calling program that the conversion code was not found.

```
CPX  #$0782     Compare X to end of table address
BNE  FZZZZZ     Not end, continue search at
                FASCII or FBAUDO
CLRA            End of tbl, ret with 'A' cleared
RTS             Or load 'A' with error indicator
```

Another method of code conversion is to form a pointer out of the character code to be converted. This pointer is then used to point to the corresponding code in a conversion table. The conversion table contains a list of the conversion codes. Each entry is located at the address in the table to which the code to be converted will point when the pointer is formed.

In the following example, the conversion from ASCII to BAUDOT is made by resetting the two most significant bits of the ASCII code to zero, thereby forming a pointer to the corresponding BAUDOT code in the conversion table. This method of setting up the pointer means the table must begin at location 00 of the page on which it resides. If it does not, a displacement constant must be added to the pointer to properly adjust it. For this routine, it is assumed that the table begins at location 00.

The conversion table uses 64 memory locations that contain the BAUDOT codes for the characters in the order corresponding to the pointer formed by the equivalent ASCII code. As in the previous look-up table, the use of substitute characters is required at the locations in the table for which no BAUDOT equivalents exist. There-

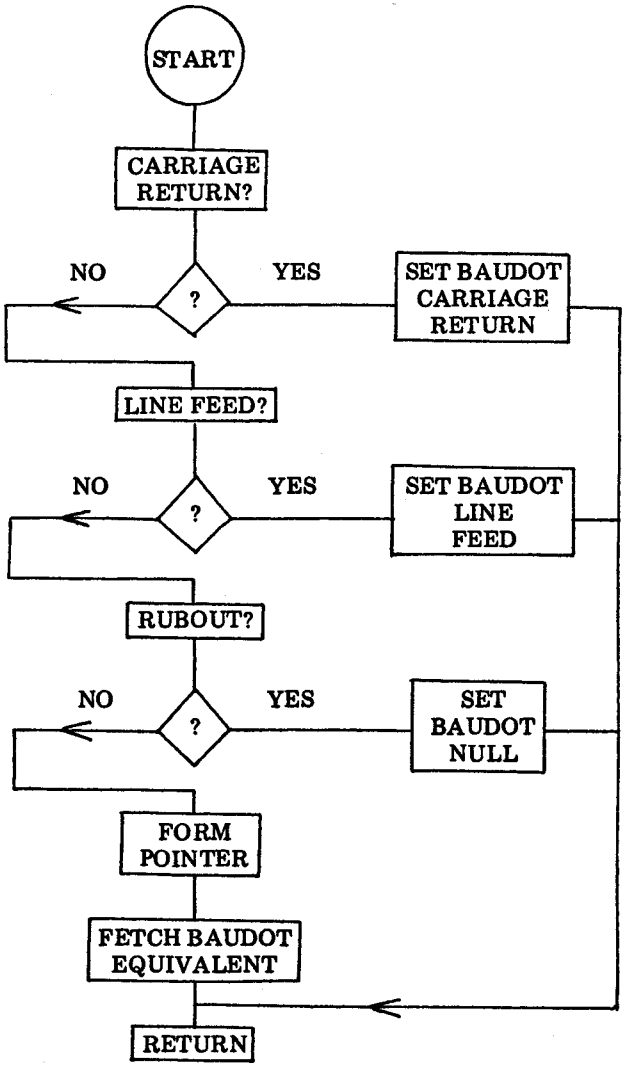
fore, the first table entry is the null character, since an @ does not exist in the BAUDOT code. The next entry is the BAUDOT code for an A, then B, and so on.

A special condition arises when the characters such as carriage return, line feed, and rubout are to be converted. In forming the pointer for these characters, it is noted that the carriage return forms a pointer to the same location as the letter M, the line feed forms a pointer to the same location as the letter J, and rubout forms a pointer to the same location as the ?. Since in this case, only three characters of this type need to be converted to BAUDOT, the conversion routine can check for their individual codes before forming the pointer. This eliminates the possibility of erroneous conversion. However, if the codes being converted have ten or more codes that overlap in this fashion, it would be more efficient (in memory usage) to expand the conversion table from 64 entries to 128, and to zero only the MSB of the ASCII code to form the pointer. This means that there will be more substitute characters contained in the table, but the actual conversion routine will not have to check each code to be converted for special characters.

The conversion routine shown next uses the pointer technique to convert from ASCII to BAUDOT, with special consideration given to the carriage return, line feed, and rubout characters. A pair of memory locations (labeled PNTR) are used to set up the pointer, which is then loaded into the index register to point to the conversion code. This routine assumes that the ASCII code of the character to be converted is contained in the A accumulator when the routine is called. The flow chart for ASBDPT is given following the listing.

```
ASBDPT   CMPA #$8D       Carriage return?
         BEQ CARRET      Yes, fetch BAUDOT carriage return
         CMPA #$8A       Line feed?
         BEQ LINFED      Yes, fetch BAUDOT line feed
         CMPA #$FF       Rubout?
         BEQ RUBOUT      Yes, fetch BAUDOT null
         ANDA #$3F       Mask off 2 MSB of ASCII
         LDAB #$04       Form pointer to conversion table
         STAB PNTR       And store in PNTR pair of
         STAA PNTR+$1    Memory locations
         LDX PNTR        Set up index register as pointer
         LDAA X          Fetch BAUDOT code from table
         RTS             Return with code in accumulator 'A'
```

```
CARRET    LDAA #$08      Set BAUDOT carriage return in 'A'
          RTS            And return

LINFED    LDAA #$02      Set BAUDOT line feed in 'A'
          RTS            And return

RUBOUT    CLRA           Set BAUDOT null in 'A'
          RTS            And return
```



0400	00	NULL FOR @
0401	03	A
0402	19	B
		INSERT BAUDOT CODES C TO Y
		FROM APPENDIX E
041A	11	Z
041B	2F	(FOR [
041C	00	NULL FOR
041D	32	) FOR]
041E	00	NULL FOR ↑
041F	00	NULL FOR ←
0420	04	SPACE
0421	2D	!
0422	31	"
0423	34	#
0424	29	$
0425	00	NULL FOR %
0426	3A	&
0427	2B	'
0428	2F	(
0429	32	)
042A	00	NULL FOR *
042B	00	NULL FOR +
042C	2C	,
042D	23	-
042E	3C	.
042F	3D	/
0430	36	0
0431	37	1
0432	33	2
0433	21	3
0434	2A	4
0435	30	5
0436	35	6
0437	27	7
0438	26	8
0439	38	9
043A	2E	:
043B	3E	;
043C	00	NULL FOR <
043D	00	NULL FOR =
043E	00	NULL FOR >
043F	39	?

There are several facts one must consider when choosing which method is to be used for code conversion. The first fact is whether the conversion will be made in both directions, from code A to code B for input, and then code B back to code A for output, or only one direction. If the conversion is only in one direction, the pointer

method would shorten the table space required since only one code is included in the table area. For conversion in both directions, either method results in approximately the same memory requirement, unless the table for the pointer method has gaps of unused locations, as would be the case if the code forming the pointer has a non-sequential bit pattern.

In programs that require speed of conversion, the pointer method is the obvious choice. It provides the correct code with just a single pass through the instruction sequence. The look-up table method will remain in a loop until the correct code is found. This means that it could take up to sixty or more times longer than the pointer method to make a single conversion.

Another common type of conversion is the conversion of numeric data from one number base to another. The typical conversion of this type is from decimal to binary, and binary to decimal. The reason this conversion is most common is that the decimal number system is used in the real world for most mathematical and numeric applications, while the computer is generally designed to operate most efficiently with binary numbers. Thus, to allow the real world and the computer to operate in their desired number systems, the conversion of decimal to binary, and vice versa, is required.

The first conversion routine converts a number designated by decimal digits in binary-coded-decimal format to the equivalent triple precision binary value. The decimal digits are contained in a table, labeled DECMAL, with one BCD digit stored per word. The table BINVAL consists of three consecutive memory locations used to store the binary number. This triple precision representation allows conversion of decimal values from 0 to 16,777,215. This routine works from the most significant decimal digit to the least significant decimal digit.

The major part of the conversion is done by a subroutine that multiplies the current contents of BINVAL by ten, and then adds one decimal digit to this new value. This subroutine, labeled TIMS10, performs the multiplication by a series of rotate and addition operations, as explained in the commented portion of the listing. The listing for this subroutine is presented next. The reader may note that several of the subroutines presented in chapter three are used by this subroutine to aid in performing its function.

The data table that preceeds this listing defines the locations used by both the binary to decimal and decimal to binary conversion routines for storing temporary data. This table indicates the number of memory bytes to be assigned to each label. The RMB in the mnemonic column is an assembler directive. It informs the assembler program of the number of bytes to be reserved for the indicated label.

```
TEMP1   RMB  $2      Temporary pointer storage
TEMP2   RMB  $2      Temporary pointer storage
DECTBL  RMB  $2      Pointer to DECMAL table
DGTCNT  RMB  $1      Counter storage for BNTODC
DECPNT  RMB  $2      Pointer to decimal constant table
BINVAL  RMB  $3      Binary equivalent storage
WRKARA  RMB  $3      Temporary working area
DECMAL  RMB  $7      Decimal equivalent storage
DECML8  RMB  $1      M.S. digit of decimal equivalent

TIMS10  PSHA         Save digit to be added
        LDX #WRKARA  Set up pointer to work area
        STX TEMP2    Save in temporary storage
        LDX #BINVAL  Set up pointer to BINVAL
        LDAB #$03    Set precision counter
        JSR MOVEIT   Move binary value to work area
        LDX #WRKARA  Set pointer to work area
        LDAB #$03    Set precision counter
        JSR ROTATL   Multiply work area X2 (total = X2)
        LDX #WRKARA  Set pointer to work area
        LDAB #$03    Set precision counter
        JSR ROTATL   Multiply work area X2 (total = X4)
        LDX #BINVAL  Set pointer to original binary value
        STX TEMP2    Save in temporary storage
        LDX #WRKARA  Set pointer to work area
        LDAB #$03    Set precision counter
        JSR ADDER    Add original to work area (total = X5)
        LDX #WRKARA  Set pointer to work area
        LDAB #$03    Set precision counter
        JSR ROTATL   Multiply work area X2 (total = X10)
        PULA         Fetch decimal digit from stack
        LDX #WRKARA  Set pointer to work area
        STX TEMP2    Store in temp storage for ADDER
        LDX #BINVAL  Set pointer to BINVAL table
        STAA X       Load binary table with decimal digit
        CLR $1,X     Clear second byte
        CLR $2,X     Clear third byte
        LDAB #$03    Set precision counter
        JSR ADDER    Add decimal digit to binary value X10
        RTS          Return with sum in BINVAL
```

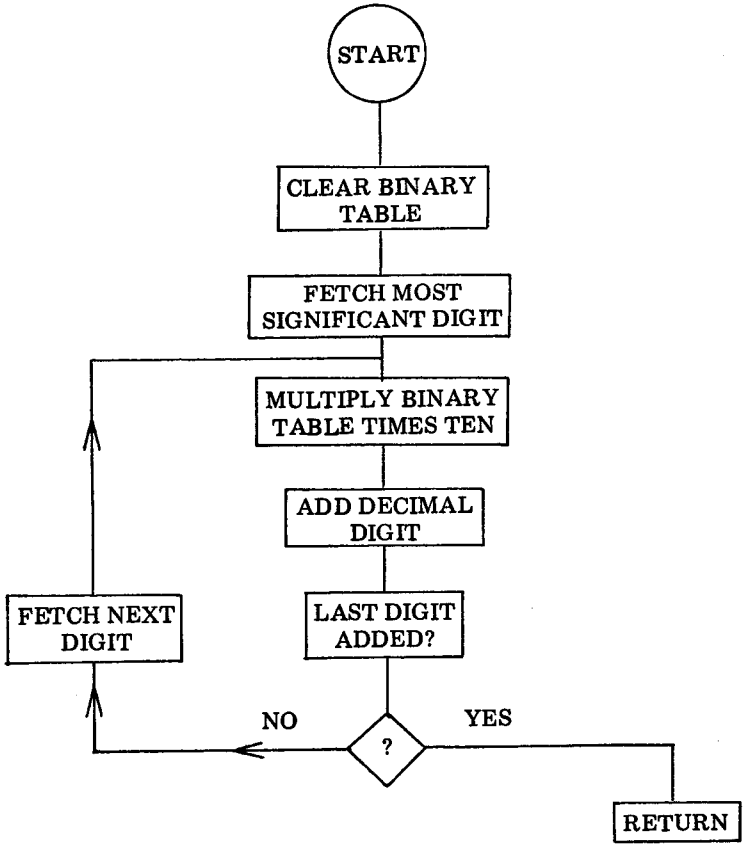
The DCTOBN routine, presented next, fetches the BCD digits from the DECMAL table for conversion to binary by the TIMS10 subroutine. First, using the CLRMEM subroutine, the three words used for the binary number storage are cleared. The routine then fetches one decimal digit at a time, beginning with the most significant digit, and calls the TIMS10 subroutine to add it to the binary value. When the conversion is complete, the routine returns to the calling program. Once again, this routine assumes the decimal digits are stored in the DECMAL table in BCD format, one digit per byte, before being called. This routine begins at the label DCTOBN.

The listing for DCTOBN is presented next, with the flow chart on the following page.

```
DCTOBN   LDX #BINVAL     Set pointer to binary value
         LDAB #$03       Set precision counter
         JSR CLRMEM      Load area with all zeros
         LDX #DECML8     Set pointer to MSD of decimal table
         STX DECTBL      Save pointer in temporary storage
DBCNVT   LDAA X          Fetch digit from decimal table
         BSR TIMS10      Add digit to binary equivalent
         LDX DECTBL      Fetch decimal table pointer
         DEX             Decrement pointer
         STX DECTBL      Restore pointer to storage
         CPX #DECMAL-$1  Is end of decimal table reached?
         BNE DBCNVT      No, continue conversion
         RTS             Otherwise, conversion is complete
```

The astute reader has undoubtably noticed that the TIMS10 subroutine may be inserted in place of the JSR TIMS10 instruction, rather than being set up as a subroutine. This has been done to call attention to the portion of the routine that performs the actual conversion. Also, as will be pointed out in the chapter on floating point arithmetic, this subroutine is used to convert the decimal numbers directly to their binary equivalents as they are entered by the operator.

The next routine performs the reverse function of the DCTOBN routine. It converts the triple precision binary value in BINVAL to the equivalent eight digit decimal number, which is then stored in the DECMAL table. This routine is called BNTODC.



BNTODC uses a subroutine labeled DCEQVL to perform the actual conversion of the binary value to decimal. The conversion is made by subtraction of a binary constant equal to a decimal power of ten. When this subroutine is called, the index register must contain the address of the least significant byte of the power of ten to be subtracted. The indicated power of ten is then subtracted from the binary value being converted until the result of the subtraction requires a borrow for the MSB. This is indicated by the carry flag being set after the subtraction is made. The SUBBER routine in chapter three is called to perform this subtraction. When the borrow occurs, the current power of ten is added back to the binary value to correct for the last subtraction. The memory location labeled DECCNT contains the decimal value for the power of ten being sub-

tracted when the subroutine returns. For example, if the binary value of one million can be subtracted five times from the binary number before the borrow occurs, the value of five would go in the seventh digit of the decimal equivalent.

This subroutine, like the TIMS10 subroutine in the previous conversion routine, can be placed in line with the BNTODC instruction sequence by replacing the BSR DCEQVL instruction with it. It is presented as a subroutine to bring out the significance of its operation to this conversion routine. The listing for the DCEQVL subroutine is given next.

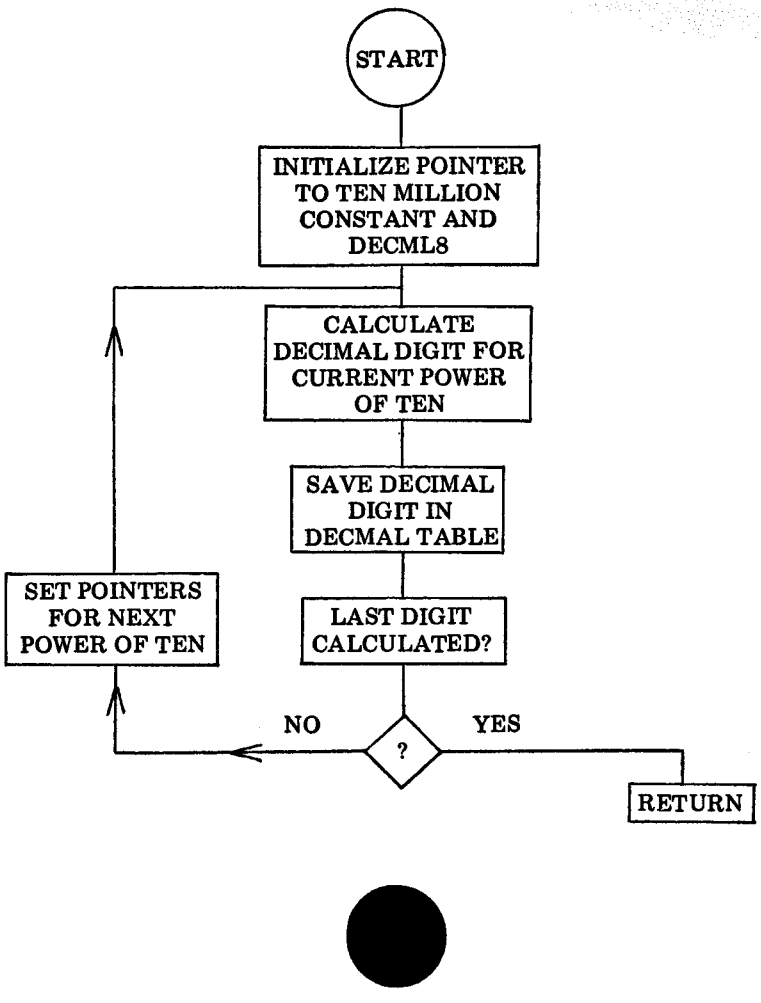
```
DCEQVL  STX TEMP2       Save binary constant pointer
        CLR DGTCNT      Zero the decimal counter
        LDAB #$03       Set precision counter
        LDX #BINVAL     Set pointer to BINVAL
DCLOOP  JSR SUBBER      Subtract dec. constant from BINVAL
        LDX DECPNT      Fetch constant pointer
        STX TEMP2       Restore pointer for add or subtract
        LDX #BINVAL     Set pointer to binary storage
        LDAB #$03       Set precision counter
        BCC INCRVL      No borrow, increment dec. counter
        JSR ADDER       Add dec. constant back to binary vlu
        RTS             Return with decimal digit in DECCNT
INCRVL  INC DGTCNT      Increment decimal counter
        BRA DCLOOP      Subtract dec. constant again
```

The BNTODC routine sets up and keeps track of the current power of ten being subtracted from the binary value by DCEQVL. As each power of ten, beginning with ten million and working down to one, is subtracted, and the value of the respective decimal digit is returned in DGTCNT, BNTODC stores the decimal digit value in DECMAL, and advances to the next lower power of ten. When the decimal value of one has been subtracted, the subroutine returns with the decimal equivalent stored in the DECMAL table. It is important to note that the value in BINVAL will be zero when the conversion is complete. The calling program must save the original value of BINVAL if it is required after the conversion.

The listing and flow chart are presented next, along with the table of binary constants for the decimal values from ten million to one.

```
BNTODC  LDX #DECML8     Set pointer to storage for
        STX DECTBL      Decimal digit storage
        LDX #TENMIL     Set pointer to start of decimal
        STX DECPNT      Constant table
BNDC    BSR DCEQVL      Calculate decimal value of digit
        LDX DECTBL      Set pointer to DECMAL table
        LDAA DGTCNT     Fetch decimal digit
        STAA X          Store digit in DECMAL table
        DEX             Back up table pointer
        STX DECTBL      Store decimal table pointer
        LDX DECPNT      Fetch decimal constant pointer
        INX             Advance pointer to next constant
        INX
        INX
        STX DECPNT      Save new decimal table pointer
        CPX #ONE+$3     Was last constant processed?
        BNE BNDC        No, continue conversion
        RTS             Yes, return, conversion complete
```

```
TENMIL  $80             Ten million in binary
        $96
        $98
ONEMIL  $40             One million in binary
        $42
        $0F
HUNTHO  $A0             One hundred thousand in binary
        $86
        $01
TENTHO  $10             Ten thousand in binary
        $27
        $00
ONETHO  $E8             One thousand in binary
        $03
        $00
HUNRED  $64             One hundred in binary
        $00
        $00
  TEN   $0A             Ten in binary
        $00
        $00
  ONE   $01             One in binary
        $00
        $00
```


START
INITIALIZE POINTER TO TEN MILLION CONSTANT AND DECML8
CALCULATE DECIMAL DIGIT FOR CURRENT POWER OF TEN
SAVE DECIMAL DIGIT IN DECMAL TABLE
LAST DIGIT CALCULATED?
NO
?
YES
SET POINTERS FOR NEXT POWER OF TEN
RETURN

FLOATING POINT ROUTINES

The concept of programming a computer to perform complex mathematical operations is often considered, by those unfamiliar with the techniques involved, to be too complicated a task to attempt. However, if one takes the time to break the program down into its basic functions, it is seen that the overall operation is not quite so difficult.

As one should be aware of by now, the digital computer is capable of performing mathematical operations with numbers of considerable magnitude. This is possible by representing numbers as multiple precision values in which more than one memory location is used to hold the numeric information. However, by increasing the number of locations assigned to represent a number, one could reach a point where the least significant bits become too insignificant with respect to the total value. A more practical representation would be to condense the size of the required number of significant digits, and indicate the overall magnitude of the value by a power of the number base. This representation is referred to as floating point format.

Floating point format allows one to define a number as a product of two values. The first value contains the significant digits of the number. This value is referred to as the "mantissa." It should contain as many significant digits as needed to properly define its relative value. The second value contains the power to which the number base is to be raised. This value, called the "exponent," indicates the magnitude of the significant digits of the mantissa. For example, the decimal value 1,000,000 would require a triple precision binary number to be properly represented. However, this same value can be defined as "1 X 10**6," or, in floating point notation, "1.0 E+6." This form contains the mantissa, 1, which is the single significant digit of this value, and the exponent, 6, which indicates the power of ten, or the number of places the decimal point should be moved to the right. One should easily realize that this shorter notation also requires fewer memory locations to represent the indicated value - one location to contain the significant digit (1), and a second to hold the exponent value (6).

One more advantage this notation has over the individual multiple precision value is the capability to represent fractional numbers. By providing a sign bit for the exponent, negative, as well as positive,

values of the exponent can be expressed. Remember, a negative exponent forms the reciprocal of the power. For base ten, the exponent, -1, would indicate the value of 1/10 times the mantissa. The negative exponent moves the decimal point to the mantissa one place to the left for each integer value of the exponent.

This notation can be used to represent binary numbers as well. The binary mantissa contains the significant bits of the binary value, and the binary exponent will indicate the power of two to which the mantissa is raised, thereby indicating the location of the decimal point (or, to properly refer to it, the binary point). The same properties of the decimal exponent apply to the exponent for the binary numbers. If the exponent is positive, the binary point in the mantissa is actually located to the right by the number of places indicated by the exponent. A negative exponent shifts the binary point to the left. Putting it in more relative terms, if the mantissa is shifted to the right, the exponent must be incremented. Shifting the mantissa to the left means the exponent must be decremented. The following illustrates three ways of expressing the same number in binary floating point format.

$$101.0\ E{+}0 = 5 \times 1 = 5$$

$$.101\ E + 3 = 5/8 \times 8 = 5$$

$$101000.0\ E - 3 = 40 \times 1/8 = 5$$

This notation may be used to represent a wide range of values with a minimum number of memory locations. One or more memory locations may be set up to store the mantissa and the exponent. The number of locations used will depend on the number of significant bits desired to express each quantity.

The floating point routines to be presented in this chapter operate with binary floating point numbers in the following format. Each number will be stored in four memory locations. The first location will contain the exponent with the most significant bit indicating the sign of the exponent. The sign bit will indicate a positive number if reset to zero, and a negative exponent if set to one. The next three locations will be used to store the mantissa as a triple precision

binary number with the most significant bit of the most significant byte used to indicate the sign of the mantissa. The binary point will always be implied to be to the right of the sign bit in the mantissa. One should note that there is no implied binary point in the exponent since the exponent is always assumed to be an integer value. This format is illustrated below.

EXPONENT	MSB	MANTISSA	LSB
SEEEEEEE	S.MMMMMMM	MMMMMMMM	MMMMMMMM
MEM LOC N+3	MEM LOC N+2	MEM LOC N+1	MEM LOC N

The order for storing the data in the memory location should be noted. The exponent is stored in the highest address of the four locations used to store the floating point number. Also, since the sign bit takes up one bit for both the mantissa and the exponent, the number of bits used to represent each value is 23 (decimal) and 7, respectively.

Before presenting the floating point routines, it should be noted that various locations on page 00 are used for data storage. This data includes pointers and counters required at different times, several temporary storage tables, and two areas that are frequently used as operating registers. These two areas shall be referred to as the floating point accumulator and the floating point operand. The floating point accumulator is used as the accumulator of the floating point routines in performing calculations and storing the results of the operations performed. The floating point operand is used to store and manipulate the number operated on by the accumulator. These two locations will have the same format as that defined for the floating point numbers just described. The floating point accumulator and operand shall be abbreviated as FPACC and FPOP throughout the remainder of this chapter.

The first routine to be presented is used to adjust the floating point numbers to a common format for proper operation of the other floating point routines. In order for the floating point arithmetic routines to operate with the highest degree of accuracy possible, the value in the FPACC must be adjusted to a standard representation before the operations are performed. This representation is referred to as the "normalized" value. A number is considered to be

normalized when the most significant bit with a value of '1' in the mantissa is in the bit position just to the right of the implied binary point. If this bit is not a '1,' the number is normalized by shifting the mantissa to the left until the most significant '1' is just to the right of the implied binary point. For each bit position shifted to the left in the mantissa, the corresponding exponent must be decremented to maintain the actual value of the number. The resultant value of the mantissa will be a number greater than or equal to one half, and less than one. This process is illustrated below:

BEFORE NORMALIZATION	0.00011011100011000011010 E+0
AFTER NORMALIZATION	0.11011100011000011010000 E-3

The process of normalizing a floating point number is required to set up the values in a common format with which the other routines can work effectively. Also, normalizing a number allows more significant digits in the mantissa. By insuring that one is using the most number of significant digits possible, the accuracy of the calculations will be increased.

The normalization routine is written to operate with positive mantissa values. If the number to be normalized is negative, this routine will convert it to its two's complement form before normalizing, and then complement it again after the normalization. The following example illustrates the process for normalizing the value -5, as it may appear after an arithmetic operation.

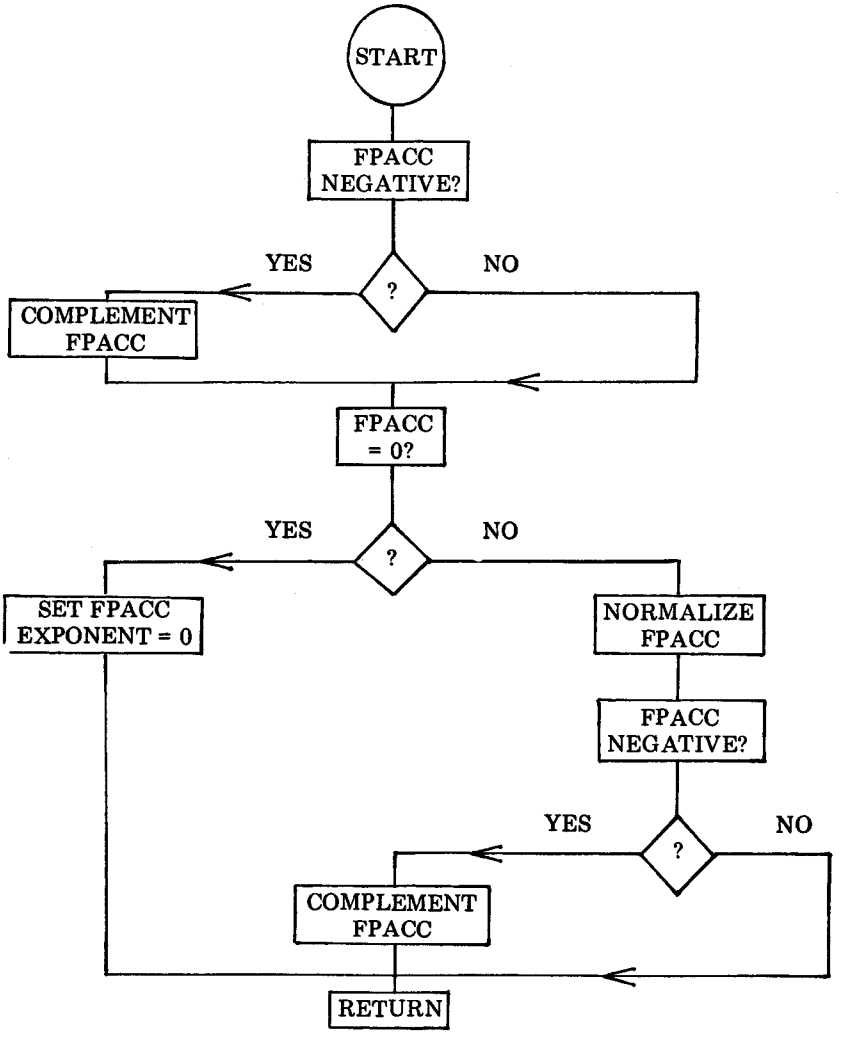
INITIAL VALUE	1.1111111 01100000 00000000	E+$0D
COMPLEMENTED	0.0000000 10100000 00000000	E+$0D
NORMALIZED	0.1010000 00000000 00000000	E+$03
COMPLEMENTED	1.0110000 00000000 00000000	E+$03

One special test that must be made by this routine is to check for an initial mantissa value of zero. If the mantissa is initially all zeros, and the normalization routine is allowed to perform its normal sequence, it would become caught in an endless loop looking for the first "one" bit. Therefore, to eliminate this possibility, the FPACC mantissa is initially checked for a value of zero, and, if found, the FPACC exponent is zeroed and the routine returns.

The listing and flow chart for the normalization routine are presented next. It should be noted that the routine uses four memory locations for the mantissa in the initial stages of the process. This is necessary to handle some special cases that occur in the multiplication routine that require the additional precision. For the routines that do not use the additional byte, the least significant byte minus one of the mantissa must be set to zero before calling the FPNORM routine.

```
FPNORM  LDX #TSIGN      Set pointer to sign register
        LDAA FPMSW      Fetch FPACC MSByte
        BMI ACCMIN      If negative, branch
        CLR X           If positive, clear sign register
        BRA ACZERT      Then branch to test if FPACC =0
ACCMIN  STAA X          Set sign indicator if minus
        LDAB #$04       Set precision counter
        LDX #FPLSWE     Set pointer to FPACC LSW-1
        JSR COMPLM      Two's complement FPACC
ACZERT  LDX #FPMSW      Set pointer to FPACC MSByte
        LDAB #$04       Set precision counter
LOOK0   TST X           See if FPACC =0
        BNE ACNONZ      Branch if non-zero
        DEX             Decrement index
        DECB            Decrement precision
        BNE LOOK0       If counter not =0, continue
        CLR FPACCE      FPACC =0, clear exponent too
NORMEX  RTS             Exit normalization subroutine
ACNONZ  LDX #FPLSWE     Set pointer to FPACC LSByte-1
        LDAB #$04       Set precision counter
        JSR ROTATL      Rotate FPACC left
        TST X           See if 1 in MSB
        BMI ACCSET      Positive - justified if so
        DEC $01,X       Else, decrement FPACC exponent
        BRA ACNONZ      Continue rotating
ACCSET  LDX #FPMSW      Set pointer to FPACC MSByte
        LDAB #$03       Set precision counter
        JSR ROTATR      Compensating rotate right FPACC
        TST TSIGN       Test original sign of FPACC
        BEQ NORMEX      Normal exit if positive sign
        LDAB #$03       With pntr at LSByte, set precision cntr
        JMP CMPLM       Restore FPACC to negative & return
```

The floating point operating program discussed in this chapter is presented in Appendix F. The operating portion of this assembled version is completely re-locatable. This means that the machine code for the operating portion of the program presented in Appendix F



may be loaded exactly as presented, without alterations, in any continuous block of memory. The only fixed memory requirement that must be adhered to for this assembled version is the locations on page 00 used to store the temporary data. (This means that page 00 must consist of RAM memory.)

In studying the listing, one may have noticed that several of the subroutines of chapter three are used in this routine. These routines

are the ROTATL, ROTATR, and COMPLM subroutines. Throughout the remainder of the floating point routines, these, and other subroutines, such as MOVEIT, CLRMEM, ADDER, and SUBBER, will be called upon to perform their various functions.

The next routine to be discussed is the floating point addition routine. The basic function of this routine is carried out by the ADDER subroutine. However, there are a number of conditions that must be considered before the actual addition is performed.

First, the FPACC and the FPOP are tested for a value of zero. If both values are zero, or only the FPOP is zero, the routine can be exited immediately, since the answer is already in the FPACC. (Remember, the results of all floating point operations are returned in the FPACC!) If the FPACC is zero, the contents of the FPOP are transferred to FPACC before returning.

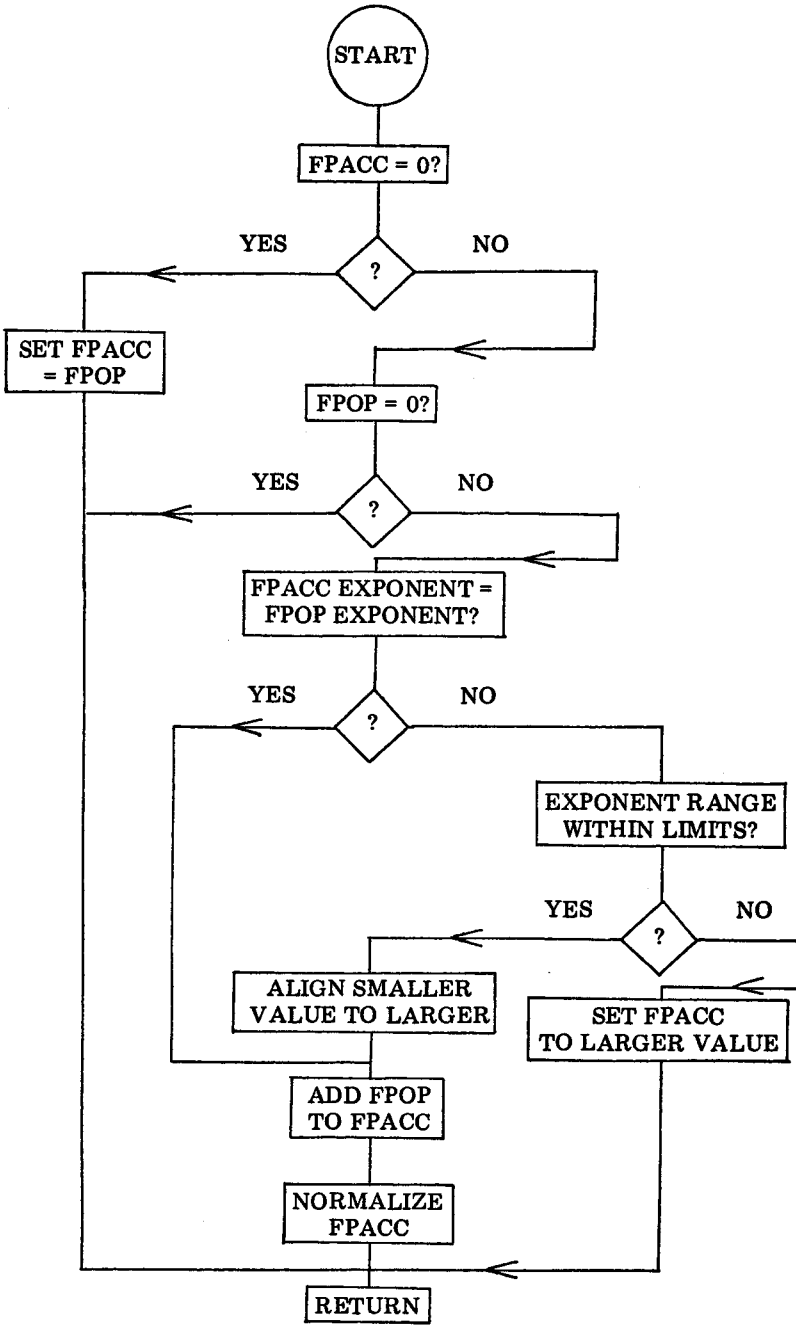
Should both numbers contain values other than zero, as is most likely the case when FPADD is called, the relative magnitude of one number to the other must be compared. With both numbers expressed in floating point notation, the range of values can vary quite a bit. For the addition routine, there is a limit in which the relative magnitude of the two numbers must fall. If one value is so much larger than the other, that the significant digits of the smaller are outside the range of the significant digits of the larger, the addition would result in no change to the larger number. The answer would simply be equal to the larger number. This range is equal to the number of bits used to represent the value of the mantissa. For the floating point format used by these routines, the allowable limit on the difference between the two exponents is seventeen (hexadecimal). If the difference is greater than seventeen, the number of greater magnitude is returned in the FPACC as the answer.

Assuming that the two numbers fall within the allowable range, the mantissas must be properly aligned before the addition can be executed. The two numbers are aligned when the exponents of each are equal. This alignment is made by shifting the mantissa of the smaller value to the right, while incrementing its exponent until it is equal to the exponent of the larger. Of course, if the exponents are equal at the start, this is not necessary. The only special consideration in this procedure is when the mantissa being shifted is negative. In this case, a '1' must be shifted into the MSB of the mantissa to maintain the negative condition. This is accomplished by

setting the carry flag and calling the second entry point, ROTR, of the ROTATR subroutine, which will not clear the carry at the start of the rotate operation.

The final operation before the addition is performed is to shift both the FPACC and the FPOP one bit to the right to maintain any overflow from the addition within the FPACC. This eliminates the necessity of checking the carry flag for an overflow condition at the end of the operation. Also, so that the least significant bits of the aligned FPACC and the FPOP are not lost as a result of this shifting operation, quad-precision is used for both the shifting and the addition. The result is normalized before returning. The flow chart and listing for FPADD are presented next.

```
FPADD   TST FPMSW       See if FPACC MSByte is equal to 0
        BNE NONZAC      Branch if not zero
MOVOP   LDX #FPLSW      Set pointer to FPACC LSByte
        STX TEMP2       Save in temporary storage
        LDX #FOPLSW     Set pointer to FPOP LSByte
        LDAB #$04       Set precision counter
        JMP MOVEIT      Move FPOP to FPACC & return

NONZAC  TST FOPMSW      See if FPOP MSByte is equal to 0
        BNE CKEQEX      Not 0, check exponents
        RTS             Exit if operand is equal to 0

CKEQEX  LDX #FPACCE     Set pointer to FPACC exponent
        LDAA X          Get FPACC exponent
        CMPA FOPEXP     Compare with FPOP exponent
        BEQ SHACOP      Branch ahead if equal
        NEGA            Else negate FPACC exponent
        ADDA FOPEXP     Add in FPOP exponent
        BPL SKPNEG      If result +, FPOP greater than FPACC
        NEGA            If result -, form two's complement
SKPNEG  CMPA #$18       See if result less than 18 hexadecimal
        BMI LINEUP      If so, can align the mantissas
        LDAB X          If not, see which is larger
        LDAA FOPEXP     Get FPOP exponent
        SBA             Subtract B from A (FPOP - FPACC)
        BPL MOVOP       If FPOP grtr than FPACC,
                        Move it to FPACC
        RTS             If FPACC grtr than FPOP, ok to exit

LINEUP  LDAA FOPEXP     Fetch FPOP exponent
        LDAB X          And FPACC exponent
        SBA             Subtract: FPOP - FPACC
        TAB             Save difference in B
```


START
FPACC = 0?
YES
?
NO
SET FPACC = FPOP
FPOP = 0?
YES
?
NO
FPACC EXPONENT = FPOP EXPONENT?
YES
?
NO
EXPONENT RANGE WITHIN LIMITS?
YES
?
NO
ALIGN SMALLER VALUE TO LARGER
SET FPACC TO LARGER VALUE
ADD FPOP TO FPACC
NORMALIZE FPACC
RETURN

```
        BMI SHIFTO      If neg., FPACC greater, shift FPOP
MORACC  LDX #FPACCE     Set pointer to FPACC exponent
        BSR SHLOOP      Branch to shift loop subroutine
        DECB            Decrement difference counter
        BNE MORACC      If not =0, then continue
        BRA SHACOP      When =0, set up for addition
SHIFTO  LDX #FOPEXP     Set pointer to FPOP exponent
        BSR SHLOOP      Branch to shift loop subroutine
        INCB            Increment difference counter
        BNE SHIFTO      Repeat loop if counter ≠0
SHACOP  CLR FPLSWE      Clear FPACC LSByte-1
        CLR FOLSWE      Clear FPOP LSByte-1
        LDX #FPACCE     Set pointer to FPACC exponent
        BSR SHLOOP      Branch to shift loop subroutine
        LDX #FOPEXP     Set pointer to FPOP exponent
        BSR SHLOOP      Do shift loop
        LDX #FOLSWE     Set pointer to FPOP LSByte-1
        STX TEMP2       Store pointer in temporary storage
        LDX #FPLSWE     Set pointer to FPACC LSByte-1
        LDAB #$04       Set precision counter
        JSR ADDER       Jump to ADDER subroutine
        JMP FPNORM      Normalize result and return

SHLOOP  INC X           Increment exponent value
        DEX             Decrement pointer
        TBA             Store difference counter
        LDAB #$04       Set precision counter

FSHIFT  TST X           Test MSB of MSByte
        BMI BRING1      If minus, shift in 1
        JSR ROTATR      Else, rotate right
        BRA RESCNT      And restore counter

BRING1  SEC             Load 1 into carry
        JSR ROTR        Do rotate with 1 in carry
RESCNT  TAB             Restore difference counter
        RTS             Return
```

Floating point subtraction may be derived by simply forming the two's complement of the value contained in the FPACC and then jumping to the FPADD routine, as the following FPSUB routine illustrates.

```
FPSUB   LDX #FPLSW      Set pointer to FPLSW
        LDAB #$03       Set precision counter
        JSR COMPLM      Complement FPACC
        JMP FPADD       Subtract by adding negative
```

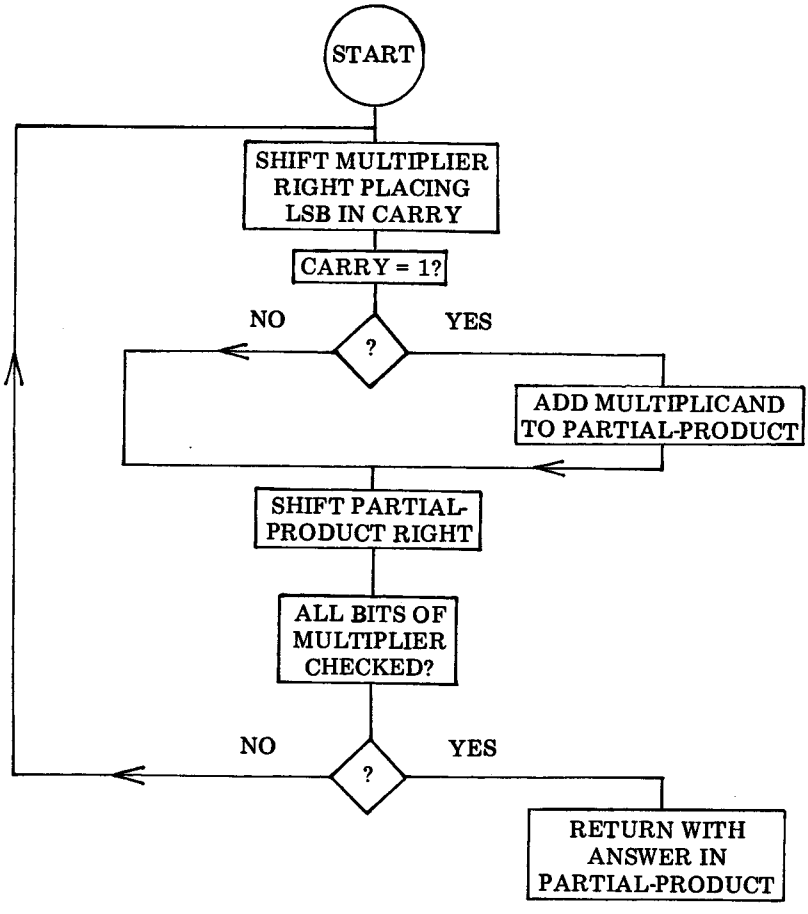
Floating point multiplication is essentially carried out by a series of shifting and addition operations. As presented previously, a binary number is multiplied by two by simply shifting it one bit position to the left. By utilizing this fact with the proper addition function, one can create a multiplication algorithm for multiple precision binary numbers. This algorithm would operate in the following manner.

The two numbers to be multiplied shall be referred to as the multiplier and the multiplicand. A third register, called the partial-product, shall be used to store the product as it is being calculated. First, examine the LSB of the multiplier. If it is a '1,' add the multiplicand to the partial-product register. After the addition, or if the LSB was zero, shift the multiplicand to the left one bit position (multiplying it by two). Examine the bit to the left of the LSB of the multiplier and, if it is a '1,' add the current value of the multiplicand to the partial-product. Then, shift the multiplicand to the left again. The process continues for each bit of the multiplier, working up to the MSB. Each time the multiplier bit is equal to '1,' the current multiplicand is added to the partial-product. The multiplicand is always shifted left following the examination of each bit of the multiplier (and addition to the partial-product if the bit is '1'). The result of the multiplication is contained in the partial-product register when the operation is complete.

The algorithm just described performs multiplication of standard binary numbers. Using this basic procedure, a multiplication algorithm for the mantissa in floating point format can be written. The following flow chart illustrates the process to be used to multiply the floating point values. The only major difference between the algorithm above and the process used by this floating point multiplication routine is that the partial-product is shifted right for each bit examined, rather than shifting the multiplicand to the left.

The exponent portion of the binary floating point numbers is manipulated in the same manner as the exponent of decimal floating point numbers for multiplication. They are simply added together.

The signs of the mantissa of both the multiplier and the multiplicand must be examined before the multiplication is executed. Since the multiplication algorithm only works for positive numbers, if either value is negative it must be two's complemented before multiplying. Also, following the laws of multiplication, if the two values are the same sign, the result will be positive; if the signs are opposite,



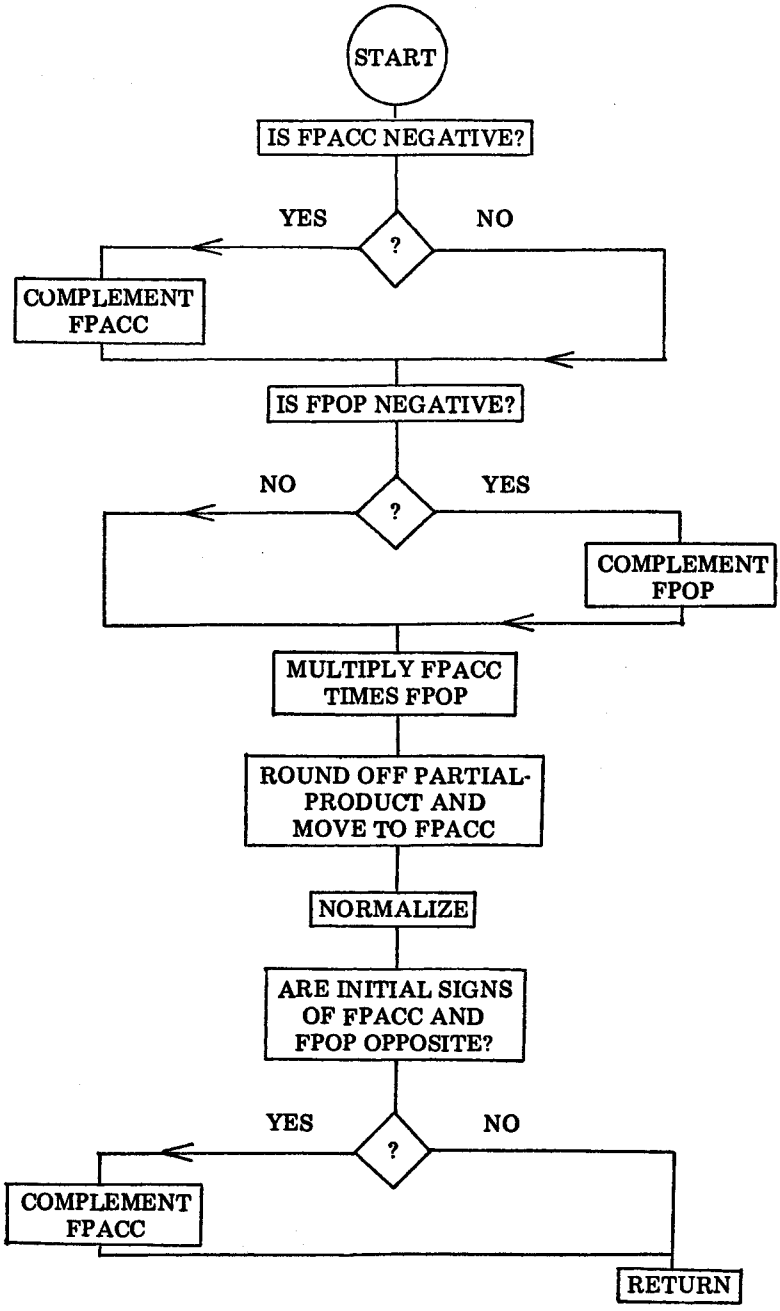
the result will be negative. This condition must be tested at the beginning, and, if the result is to be negative, the final value must be two's complemented before returning.

It should also be noted that if the partial-product is rotated right for each bit of the multiplier, it is necessary for the partial-product register to contain twice as many bits as the multiplier. Although the partial-product register contains more precision than the program is designed to handle, it is necessary to maintain the required significant bits for the answer. At the completion of the multiplication algorithm, the 24th bit of the partial-product is used to round off the final result, which is then normalized to the proper 23 bit floating

point format. This manner of handling the partial-product allows maximum precision for the multiplication routine.

The flow chart and listing for the FPMULT routine are now presented.

```
FPMULT  BSR CKSIGN      Set up and check sign of numbers
ADDEXP  LDAA FOPEXP     Get FPOP exponent
        ADDA FPACCE     Add FPACC exponent
        INCA            Add 1 for algorithm compensation
        STAA FPACCE     Store result in FPACC exponent
SETMCT  LDAA #$17       Set 17 hex (23 dec) as bit counter
        STAA CNTR       Store bit counter
MULTIP  LDX #FPMSW      Set pointer to FPACC MS Byte
        LDAB #$03       Set precision counter
        JSR ROTATR      Rotate FPACC right
        BCC NADOPP      Carry =0, don't add to partial-product
ADOPPP  LDX #MCAND1     Pointer to LSByte of multiplicand
        STX TEMP2       Store pointer
        LDX #WORK1      Pointer to LSBtye of partial-product
        LDAB #$06       Set precision counter
        JSR ADDER       Perform addition
NADOPP  LDX #WORK6      Set pntr to MSByte of partial-product
        LDAB #$06       Set precision counter
        JSR ROTATR      Rotate partial-product right
        DEC CNTR        Decrement bit counter
        BNE MULTIP      Continue multiplying if not zero
        LDX #WORK6      Else, set pointer to partial-product
        LDAB #$06       Set precision counter
        JSR ROTATR      Make room for possible rounding
        LDX #WORK3      Set pntr to 24th bit of partial-product
        LDAA X          Fetch LS Byte-1 of result
        ROLA            Rotate 24th bit to sign
        BPL PREXFR      If 24th bit =0, branch ahead
        LDAB #$03       Set precision counter
        LDAA #$40       Add 1 to 23rd bit of partial-product
        ADDA X          To round off result
CROUND  STAA X          Restore to memory
        INX             Increment pointer
        LDAA #$00       Clear A without changing carry
        ADCA X          Add with carry to propogate
        DECB            Any rounding - decrement counter
        BNE CROUND      Continue until counter =0
        STAA X          Restore last byte to memory
PREXFR  LDX #FPLSWE     Set pointer to FPACC LSW-1
        STX TEMP2       Store pointer
        LDX #WORK3      Set pointer to LSW of working register
        LDAB #$04       Set precision counter
```


START
IS FPACC NEGATIVE?
YES
?
NO
COMPLEMENT FPACC
IS FPOP NEGATIVE?
NO
?
YES
COMPLEMENT FPOP
MULTIPLY FPACC TIMES FPOP
ROUND OFF PARTIAL-PRODUCT AND MOVE TO FPACC
NORMALIZE
ARE INITIAL SIGNS OF FPACC AND FPOP OPPOSITE?
YES
?
NO
COMPLEMENT FPACC
RETURN

```
EXMLDV  JSR MOVEIT      Move partial-product to FPACC
        JSR FPNORM      Normalize result
        LDAA SIGNS      Get sign storage
        BNE MULTEX      If sign indicator ≠ 0, sign is positive
        LDX #FPLSW      Else pointer to FPACC LSByte
        LDAB #$03       Set precision counter
        JSR COMPLM      Complement result
MULTEX  RTS             Exit FPMULT subroutine

CKSIGN  LDAB #$08       Set counter in B
        LDX #WORK0      Set pntr to partial-product work area
CLMOR1  CLR X           Fill memory with zeros
        INX             Advance pointer
        DECB            Decrement counter
        BNE CLMOR1      Not done, do more
        LDAB #$04       Reset counter
        LDX #MCAND0     Set pointer to multiplicand storage
CLMOR2  CLR X           Put zeros in memory
        INX             Advance pointer
        DECB            Decrement counter
        BNE CLMOR2      Repeat until counter =0
        LDAA #$01       Put 1 into signs
        STAA SIGNS      Indicator to initialize
        LDAA FPMSW      Fetch FPACC MS Byte
        BPL OPSGNT      Minus? Complement value
NEGFPA  DEC SIGNS       Decrement signs indicator
        LDX #FPLSW      Set pointer to FPACC LSByte
        LDAB #$03       Set precision counter
        JSR COMPLM      Two's complement FPACC
OPSGNT  TST FOPMSW      FPOP MSB =1?
        BMI NEGOP       Minus? Complement value
        RTS             Else, return

NEGOP   DEC SIGNS       Decrement signs indicator
        LDX #FOPLSW     Set pointer to FPOP LSByte
        LDAB #$03       Set precision counter
        JMP COMPLM      Two's complement FPOP & return
```

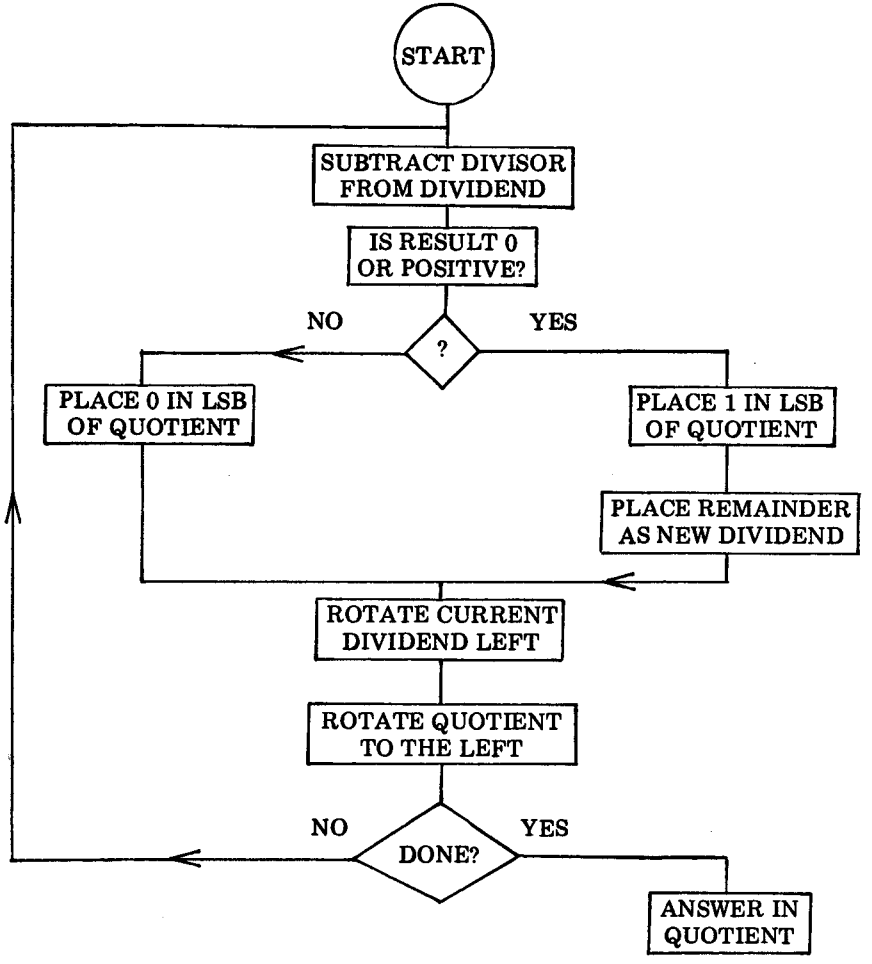
The procedure for division can almost be considered the reverse of that for multiplication. The division algorithm consists of a series of subtraction and shifting operations. This algorithm is illustrated in the following flow chart. This algorithm is written for division of numbers in floating point format rather than straight binary. For operating with numbers in standard binary format, the most significant bits of the divisor and dividend would have to be properly aligned, and the location of the binary point in the quotient would have to be accounted for in cases where the result is not a pure integer.

Rather than verbalize the operation as presented in the flow chart, a sample division of two floating point numbers using this algorithm will be presented in a step-by-step fashion. This illustration will divide the binary equivalent of the value 15 (decimal) by 5. The numbers are presented as 4 bit values to keep the illustration short. However, in the FPDIV routine, the operation is carried out 23 times for each significant bit of the mantissa of the dividend. Once again, this algorithm assumes the numbers are in normalized floating point format.

```
0.1111      Original DIVIDEND at start of routine.

0.1010      DIVISOR (Note floating point format.)
--------
0.0101      This is the REMAINDER from the subtraction
            operation. Since the result was POSITIVE, a '1'
            is placed in the LSB of the QUOTIENT register.

            0.0001  QUOTIENT after 1'st loop.

NOW BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER)
ARE ROTATED LEFT

0.1010      New DIVIDEND (which is the previous REMAIN-
            DER rotated once to the LEFT).
0.1010      DIVISOR (Does not change during routine).
--------
0.0000      RESULT of this subtraction is zero and thus
            qualifies to become a NEW DIVIDEND. Also,
            QUOTIENT LSB gets a '1' for this case!

            0.0011  QUOTIENT after 2'nd loop

AGAIN BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER)
ARE ROTATED LEFT

0.0000      New DIVIDEND (which is the last remainder
            rotated once to the left).
0.1010      DIVISOR (still same old number).
--------
1.0110      RESULT of this subtraction is a minus number
            (note that the SIGN bit changed). Thus, old
            DIVIDEND stays in place and QUOTIENT
            gets a '0' in LSB!

            0.0110  QUOTIENT after 3'rd loop
```

NOW BOTH QUOTIENT, AND IN THIS CASE, THE OLD DIVIDEND, ARE ROTATED LEFT

0. 0 0 0 0	Old DIVIDEND rotated once to the left.
0. 1 0 1 0	Same old DIVISOR
1. 0 1 1 0	RESULT of this subtraction is again a minus. Old DIVIDEND stays in place. QUOTIENT gets another '0' in LSB.
0. 1 1 0 0	QUOTIENT after 4'th loop



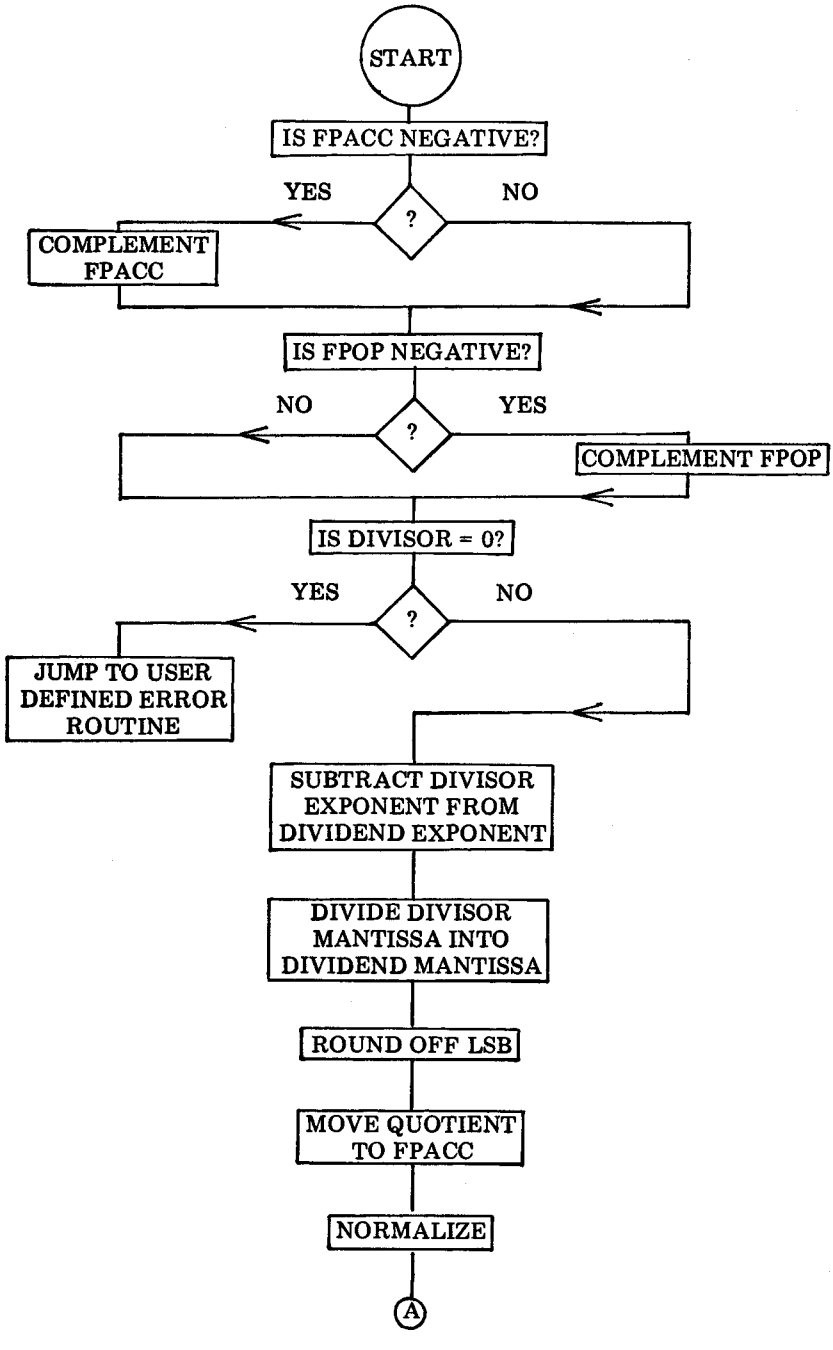
With only four significant bits in the dividend, the calculation illustrated ends after the fourth loop. The answer is contained in the quotient. The exponents are the next quantity that must be dealt with, since the values are represented in floating point notation. Just as in division of decimal floating point numbers, the exponents of the binary counterparts are subtracted, DIVIDEND exponent minus the DIVISOR exponent. In the example given, the dividend would have an exponent of four for the normalized binary value of 15 (decimal), and the divisor would have a binary exponent of three. The algorithm as presented requires a compensation factor of +1 after subtracting the exponents in order to have the correct floating point result. Thus, the exponent of the quotient in the previous example would be (4-3)+1=2. This can be verified by moving the implied binary point in the quotient two places to the right - the binary value of 3 would indeed be observed.

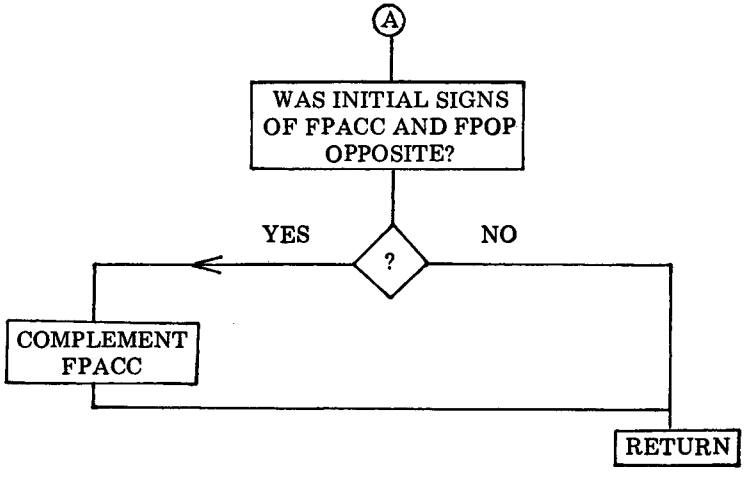
In the division algorithm, just as in the multiplication, the sign of the dividend and divisor must be positive for the algorithm to operate properly. If either is negative, it must be two's complemented before the division is performed. Also, if the signs are the same, the sign of the quotient must be positive. If the signs are opposite, the quotient must be two's complemented before exiting the routine to make the answer negative.

The FPDIV routine is listed next along with the flow chart. As one examines the listing, one should observe that two other conditions are considered by the routine. If the quotient has a remainder after the final loop through the divide algorithm, which would result in a '1' in the 24th bit position, the quotient will be rounded off by adding a '1' to the 23rd bit. Also, if a divide by zero is attempted (which is an illegal operation), the FPDIV routine jumps to a routine labeled DERROR. This routine, as listed, outputs a "?" via the MIKBUG** output routine, and then returns to the calling program. The user may revise this to perform whatever is deemed necessary when this error occurs.

```
FPDIV   JSR CKSIGN      Clear work area and set up sign
        TST FPMSW       Check for divide by 0
        BEQ DERROR      DIVISOR =0, divide by zero error

SUBEXP  LDAA FOPEXP     Get DIVIDEND exponent
        SUBA FPACCE     Subtract DIVISOR exponent
        INCA            Compensate for divide algorithm
        STAA FPACCE     Store result in FPACC exponent
```

```
SETDCT  LDAA #$17       Set bit counter storage
        STAA CNTR       To 17 hexadecimal (23 decimal)

DIVIDE  BSR SETSUB      Subtract DIVISOR from DIVIDEND
        BMI NOGO        If result is minus, 0 in QUOTIENT
        LDX #FOPLSW     TO location for MOVEIT
        STX TEMP2       Store pointer
        LDX #WORK0      FROM location in pointer
        LDAB #$03       Set precision counter
        JSR MOVEIT      Move DIVIDEND from
                        Work area to FPOP
        SEC             Set carry for positive results
        BRA QUOROT      Rotate into QUOTIENT

DERROR  LDAA #$BF       Set ASCII for "?"
        JMP $E1D1       Print "?" and return

NOGO    CLC             Negative result, clear carry

QUOROT  LDX #WORK4      Set pointer to LSByte of QUOTIENT
        LDAB #$03       Set precision counter
        JSR ROTL        Rotate carry into LSB of QUOTIENT
        LDX #FOPLSW     Set pointer to DIVIDEND LSByte
        LDAB #$03       Set precision counter
        JSR ROTATL      Rotate DIVIDEND left
        DEC CNTR        Decrement bit counter
        BNE DIVIDE      If not finished, continue
        BSR SETSUB      Do one more for rounding
        BMI DVEXIT      If minus, no rounding
        LDAA #$01       If 0 or +, add 1 to 23rd bit
        ADDA WORK4      QUOTIENT to round off
        STAA WORK4      Restore LSByte of QUOTIENT
        LDAA #$00       Clear accumulator, not carry
        ADCA WORK5      Add carry to NSByte QUOTIENT
        STAA WORK5      Return to memory
        LDAA #$00       Clear accumulator, not carry
        ADCA WORK6      Add carry to MSByte QUOTIENT
        STAA WORK6      Store results
        BPL DVEXIT      If MSB of MSByte =0, exit
        LDX #WORK6      Else prepare to rotate right
        LDAB #$03       Set precision counter
        JSR ROTATR      Clear sign bit move right
        INC FPACCE      Compensate for rotate

DVEXIT  LDX #FPLSWE     Prepare to move
        STX TEMP2       QUOTIENT to FPACC
        LDX #WORK3      Set pointer to QUOTIENT
        LDAB #$04       Set precision counter
        JMP EXMLDV      Exit through FPMULT routine

SETSUB  LDX #WORK0      Move DIVISOR to
```


START
IS FPACC NEGATIVE?
YES
?
NO
COMPLEMENT FPACC
IS FPOP NEGATIVE?
NO
?
YES
COMPLEMENT FPOP
IS DIVISOR = 0?
YES
?
NO
JUMP TO USER DEFINED ERROR ROUTINE
SUBTRACT DIVISOR EXPONENT FROM DIVIDEND EXPONENT
DIVIDE DIVISOR MANTISSA INTO DIVIDEND MANTISSA
ROUND OFF LSB
MOVE QUOTIENT TO FPACC
NORMALIZE
A




```
        STX TEMP2       Work area
        LDX #FPLSW      Set pointer to FPACC
        LDAB #$03       Set precision counter
        JSR MOVEIT      Move FPACC to working register
        LDX #WORK0      Prepare for subtraction
        STX TEMP2       Store pointer to DIVISOR
        LDX #FOPLSW     Set pointer to FPOP LS Byte-1
                          (DIVIDEND)
        LDAB #$03       Set precision counter
SUBBER  CLC             Clear carry flag
        LDAA X          Fetch FPOP byte (DIVIDEND)
        STX TEMP1       Store FPOP pointer
        LDX TEMP2       Fetch pointer to work area
        SBCA X          Subtract DIVISOR from DIVIDEND
        STAA X          Store result in work area
        INX             Advance work area pointer
        STX TEMP2       Store work area pointer
        LDX TEMP1       Fetch pointer to FPOP
        INX             Advance pointer to FPOP
        DECB            Decrement precision counter
        BNE SUBBER+$1   Not 0, continue subtraction
        TST WORK2       Set sign bit result in N flag
        RTS             Return with flags set
```

The floating point routines presented to this point, when assembled into the object code, will reside in approximately two and a half pages of memory. Additional memory is required for the data areas on page 0 which are used to store various counters and data values.

The locations used on page 00 by these floating point routines are listed in the following table. The addresses listed here are the same as those used by the relocatable floating point package presented in Appendix F.

ADDRESS	PROGRAM LABEL	DEFINITION
0020	CNTR	Counter Storage
0021	TEMP1	Temporary Pointer Storage
0023	TEMP2	Temporary Pointer Storage
0025	TSIGN	Sign Indicator
0026	SIGNS	Signs Indicator (Multiply & Divide)
0027	FPLSWE	FPACC Extension
0028	FPLSW	FPACC Least Significant Byte
0029	FPNSW	FPACC Next Significant Byte
002A	FPMSW	FPACC Most Significant Byte
002B	FPACCE	FPACC Exponent
002C	MCAND0	Multiplication Work Area
002D	MCAND1	Multiplication Work Area
002E	MCAND2	Multiplication Work Area
002F	FOLSWE	FPOP Extension
0030	FOPLSW	FPOP Least Significant Byte
0031	FOPNSW	FPOP Next Significant Byte
0032	FOPMSW	FPOP Most Significant Byte
0033	FOPEXP	FPOP Exponent
0034	WORK0	Work Area
0035	WORK1	Work Area
0036	WORK2	Work Area
0037	WORK3	Work Area
0038	WORK4	Work Area
0039	WORK5	Work Area
003A	WORK6	Work Area
003B	WORK7	Work Area

The floating point routines just presented are extremely powerful routines that can be of considerable value to someone who requires such mathematical calculations on a 6800 based microcomputer. These routines provide the capability to handle binary numbers equivalent to six or seven significant decimal digits raised to plus or minus the 38th power of ten. Using these routines as a base, a wide variety of mathematic operations can be performed by loading FPACC and FPOP with the numbers in normalized floating point format and calling the proper routine. By developing programs that make a series of calculations in this manner, one can solve quite

sophisticated equations, such as the expansion series for calculating the sine and cosine functions.

The range of values encompassed by these routines, as presented, may well be within the requirements of most applications. However, should it be desired to increase the number of significant digits by expanding the precision of the mantissa, or to extend the exponent to a double or triple precision value, the interested programmer should have little difficulty in making the required modification to these routines.

As pointed out in the chapter on conversion routines, one of the most common requirements of a program that deals with binary numbers is the conversion to and from decimal. This is because it is most often necessary to communicate with a human operator. And, since it is inevitable that the operator is most familiar with expressing numbers as decimal values, the conversion must be made. Therefore, to illustrate a method for converting from floating point decimal to floating point binary, and then back to floating point decimal, and also to provide a complete floating point mathematical program, the following three routines are included.

The first of these routines performs the conversion of decimal floating point numbers to floating point binary. The overall requirement of this routine is to receive the decimal number in floating point format, normalize the mantissa portion to an all integer value, and convert to the equivalent floating point binary value.

Floating point decimal values may be expressed in various forms, as indicated below.

123.45
or
1.2345 E+2

As either of these formats is received, the mantissa portion is converted to binary, while keeping track of the exponent to provide the proper normalized decimal value. Unlike the binary normalization, which shifts the binary point to the left of the MSB to provide a purely fractional mantissa, decimal normalization maintains the decimal point to the right of the least significant digit to provide a purely

integer mantissa. Thus, the example above would be normalized to:

12345 E-2

The conversion of the decimal mantissa to binary is accomplished by the routine labeled DECBIN, which is a version of the TIMS10 subroutine presented in chapter four. The listing for DECBIN will be presented with the floating point output routine, where it is also used. As indicated in chapter four, this subroutine converts each digit by first multiplying the binary equivalent of the digits already received by ten, and adding the BCD value of the latest digit input to the new binary value.

Once the mantissa is converted, the decimal exponent is input and converted to binary. At this point, it is necessary to normalize the mantissa of the binary equivalent by calling the FPNORM routine. The FPACC exponent is set to a value of 17 (hexadecimal) before calling the FPNORM routine. Then, using the FPMULT routine, the normalized binary equivalent is multiplied by 10 (for each unit of a positive decimal exponent received), or by 0.1 (for each unit of a negative exponent received).

The input and output portions of this routine require that the user provide driver routines for the specific input and output devices associated with one's system. The requirement for the INPUT routine is to return to the calling routine with the ASCII code for the character entered on the input device, such as an electronic keyboard, in the A accumulator. The routine to output characters to a display device, such as a mechanical printer or video display, must accept the character to be output as an ASCII character stored in the A accumulator. This output routine, labeled ECHO, is called to "echo" the characters received from the input device back to the display device. The reader may refer to the chapter on I/O processing for methods of creating these routines. The ECHO routine should return with the ASCII code for the character outputted in the A accumulator. (Versions of INPUT and ECHO that utilize the MIKBUG** I/O routines are presented immediately following the input routine listing.)

The decimal to binary input routine listing is presented next. One should note in the listing that both formats illustrated previously are allowed as legal entries, and the routine accounts for positive and

negative mantissas and exponents. Also, the operator has the option to cancel the current input by entering a control O character. Several locations on page 00 are used to store the input characters and save counters and indicators. These locations will be summarized later in this chapter.

```
FPINP   LDX #INMTAS     Set pointer to storage area
        LDAB #$0C       Set up counter
CLRNXT  CLR X           Clear storage area
        INX             Advance pointer
        DECB            End of clear?
        BNE CLRNXT      No, clear more
        JSR INPUT       Get character from I/O
        CMPA #$AB       Test if + sign
        BEQ SECHO       Yes, echo and continue
        CMPA #$AD       Test if - sign
        BNE NOTPLM      No, test if valid character
        STAA INMTAS     Make input sign non-zero
SECHO   JSR ECHO        Echo character to I/O
NINPUT  JSR INPUT       Get next character from I/O
NOTPLM  CMPA #$8F       Test if CONTROL O
        BNE SERASE      No, skip erase
ERASE   LDAA #$BC       Yes, ASCII code for <
        JSR ECHO        Output < to indicate deletion
        JSR SPACES      Output a few spaces
        BRA FPINP       Restart input string
SERASE  CMPA #$AE       Test if period (.)
        BNE SPRIOD      No, skip period
PERIOD  TST INPRDI      Test for period already received
        BEQ PER1        No period received yet, continue
        JMP ENDINP      Else, end input
PER1    CLR CNTR        Else, reset digit counter
        STAA INPRDI     Set (.) indicator
        JSR ECHO        Echo (.) to I/O
        BRA NINPUT      Get next character
SPRIOD  CMPA #$C5       Test if E for exponent
        BNE SFNDXP      No, skip exponent
FNDEXP  JSR ECHO        Yes, echo E to I/O
        JSR INPUT       Input next part of exponent
        CMPA #$AB       Test if + sign
        BEQ EXECHO      Yes, echo it
        CMPA #$AD       Test if - sign
        BNE NOEXPS      No, see if valid character
        STAA INEXPS     Yes, store as minus indicator
EXECHO  JSR ECHO        Echo to I/O
EXPINP  JSR INPUT       Get next character for exponent
NOEXPS  CMPA #$8F       Test if CONTROL O
        BEQ ERASE       Yes, reenter string
        CMPA #$B0       Number, test lower limit
```

```
          BMI ENDINP      No, end input string
          CMPA #$BA       Test upper limit
          BPL ENDINP      No, end input string
          ANDA #$0F       Mask and strip ASCII
          TAB             Store pure BCD in register B
          LDX #IOEXPD     Set pointer to exponent storage
          LDAA #$03       Test for upper limit of exponent
          CMPA X          Is ten's digit greater than 3?
          BMI ENDINP      Yes, end input
          LDAA X          Store temporarily in A
          CLC             Clear carry bit
          ROL X           Exponent x2
          ROL X           Exponent x4
          ADDA X          Add original (total = x5)
          ROLA            Exponent x10
          ABA             Add new number
          STAA X          Store in exponent storage
          LDAA #$B0       Restore ASCII code
          ABA             By adding B0
          BRA EXECHO      Echo number, look for next
SFNDXP    CMPA #$B0       Test if valid number
          BMI ENDINP      If too low, end input
          CMPA #$BA       Test upper limit
          BPL ENDINP      If not valid, end input
          TAB             Store in temporary register
          LDAA #$F8       Input too large?
          BITA IOSTR2     Test if so
          BNE NINPUT      Yes, ignore present input
          'BA             O.K., fetch digit from temp storage
          JSR ECHO        Echo to I/O
          INC CNTR        Increment digit counter

          ANDB #$0F       Mask off ASCII
          PSHB            Store BCD in temporary storage
          JSR DECBIN      Multiply previous value x10
          LDX #IOLSW      Set pointer to work area
          CLR $2,X        Clear MS Byte work area
          CLR $1,X        Clear NS Byte work area
          PULA            Fetch last BCD number
          STAA X          Store LS Byte in work area
          STX TEMP2       Save pointer to work area
          LDX #IOSTR      Set pointer to storage area
          LDAB #$03       Set precision counter
          JSR ADDER       Add latest number
          JMP NINPUT      Look for next character

ENDINP    TST INMTAS      Test if positive or negative
          BEQ FINPUT      Indicator zero, number positive
          LDX #IOSTR      Index to LSByte input mantissa
          LDAB #$03       Set precision counter
```

```
        JSR COMPLM      Two's complement to negate number
FINPUT  CLR IOSTR-$1    Clear input storage LSByte -1
        LDX #FPLSWE     Set pointer to FPACC LSByte -1
        STX TEMP2       Store pointer
        LDX #IOSTR-$1   Set pointer to storage LSByte -1
        LDAB #$04       Set precision counter
        JSR MOVEIT      Move input to FPACC
        LDAB #$17       Set exponent for FPNORM operation
        STAB FPACCE     Store exponent for normalization
        JSR FPNORM      Normalize input
        LDAA INEXPS     Test exponent sign indicator
        BEQ POSEXP      Positive? Same exponent
        NEG IOEXPD      Minus, make negative
POSEXP  LDAA INPRDI     Test period indicator
        BEQ EXPOK       If nothing, no decimal point
        CLRA            Clear accumulator A
        SUBA CNTR       Counter -0 to form negative
EXPOK   ADDA IOEXPD     Add to compensate for decimal point
        STAA IOEXPD     Store results
        BMI MINEXP      If value was minus, branch
        BNE EXPFIX      If plus, not finished
        RTS             If zero, return, value in FPACC

EXPFIX  BSR FPX10       Multiply FPACC x10 until
        BNE EXPFIX      Exponent is zero
        RTS             Exit, converted value in FPACC

FPX10   LDAA #$04       Place 10 decimal in FPOP by
        STAA FOPEXP     Settin FPOP exponent to 4
        LDAA #$50       And loading mantissa with 50 00 00
        STAA FOPMSW
        CLRA
        STAA FOPNSW
        STAA FOPLSW
        JSR FPMULT      Multiply FPACC x10
        DEC IOEXPD      Decrement decimal exponent value
        RTS             Return to calling program

MINEXP  BSR FPD10       Compensated decimal exponent minus
        BNE MINEXP      FPACC x0.1 till decimal exponent =0
        RTS             Return

FPD10   LDAA #$FD       Place 0.1 in FPOP by
        STAA FOPEXP     Setting FPOP exponent to -3
        LDAA #$66       And loading mantissa with 66 66 67
        STAA FOPMSW
        STAA FOPNSW
        LDAA #$67
        STAA FOPLSW
        JSR FPMULT      FPACC x0.1
        INC IOEXPD      Increment decimal exponent and
        RTS             Return to calling program
```

```
SPACES  LDAA #$A0       ASCII "space"
        JSR ECHO        Output space character
                        Fall through to ECHO for 2nd space

 ECHO   PSHA            Save character being output
        JSR $E1D1       Output character through MIKBUG**
        PULA            Restore character output
        RTS

INPUT   LDAA #$3C       Out. reset of echo for MIKBUG** rtn
        STAA $8007      Output reset code to TTY interface
        JSR $E1AC       Input char through MIKBUG** rtn
        ORAA #$80       Set MSB of ASCII code
        RTS
```

The next routine converts the floating point binary number in the FPACC to its floating point decimal equivalent, and outputs it to the display device as ASCII characters in the following format:

0.1234567 E+07

First, the normalized binary value is converted to a binary value in which the binary exponent is within the range of -4 to -1. As this is done, the decimal exponent is generated. Once the binary exponent is properly adjusted, the decimal mantissa is output by multiplying the adjusted binary mantissa by ten for each decimal digit. Each multiplication causes the next decimal digit to be pushed out into the most significant byte+1 of the binary mantissa in its BCD code. As each digit is pushed out, its ASCII code is formed, and the ECHO routine is called to output the digit. When the mantissa has been output, the decimal exponent is converted, using the method described in chapter four for binary to decimal conversion, and then output.

The listing for the FPOUT routine is presented next.

```
 FPOUT  CLR IOEXPD      Clear decimal exponent storage
        TST FPMSW       Is number to be output negative?
        BMI OUTNEG      Yes, make positive and output '-'
        LDAA #$AB       Else, set ASCII code for '+'
        BRA AHEAD1      Go display + sign

OUTNEG  LDX #FPLSW      Set pointer to LSByte of FPACC
        LDAB #$03       Set precision counter
```

```
        JSR COMPLM      Make FPACC positive
        LDAA #$AD       Set ASCII code for - sign
AHEAD1  JSR ECHO        Display sign of mantissa
        LDAA #$B0       Set ASCII 0
        JSR ECHO        Display the character
        LDAA #$AE       Set ASCII (.)
        JSR ECHO        Display it
        DEC FPACCE      Decrement FPACC exponent
DECEXT  BPL DECEXD      If compensated, exponent grtr than or
                        Equal to 0, multiply mantissa by 0.1
        LDAA #$04       Exponent is negative add 4 (dec.) to
        ADDA FPACCE     Exponent value
        BPL DECOUT      If exponent > =0, output mantissa
        JSR FPX10       Else, multiply mantissa by 10
DECREP  LDAA FPACCE     Get exponent
        BRA DECEXT      Repeat above test, > =0

DECEXD  JSR FPD10       Multiply FPACC by 0.1
        BRA DECREP      Check status of FPACC exponent
DECOUT  LDX #IOSTR      Set up for move operation
        STX TEMP2       Store pointer to working register
        LDX #FPLSW      Set pointer to FPACC
        LDAB #$03       Set precision counter
        JSR MOVEIT      Move FPACC to output registers
        CLR IOSTR3      Clear out register MSByte +1
        LDX #IOSTR      Set pointer to LSByte output register
        LDAB #$03       Set precision counter
        JSR ROTATL      Rotate to compensate for sign bit
        JSR DECBIN      Output reg x10 overflow in MSByte+1

COMPEN  INC FPACCE      Increment FPACC exponent
        BEQ OUTDIG      Go output digits when comp. done
        LDX #IOSTR3     Else, rotate right to compensate for
        LDAB #$04       Any remainder in binary exponent
        JSR ROTATR      Perform rotate right operation
        BRA COMPEN      Repeat loop until exponent = 0

OUTDIG  LDAA #$07       Set digit counter to 7
        STAA CNTR       For output operation
        LDAA IOSTR3     Fetch BCD, see if first digit =0
        BEQ ZERODG      First digit =0? Yes, branch
OUTDGS  LDAA IOSTR3     Get BCD from output register
        ADDA #$B0       Form ASCII code by adding B0
        JSR ECHO        And output digit
DECRDG  DEC CNTR        Decrement digit counter
        BEQ EXPOUT      Equal to 0, done, output exponent
        BSR DECBIN      Else, get next digit
        BRA OUTDGS      Form ASCII and output

ZERODG  DEC IOEXPD      Decr exponent for skipping display
        TST IOSTR2      Check if entire mantissa =0
```

```
          BNE DECRDG      Not 0, continue output sequence
          TST IOSTR1
          BNE DECRDG
          TST IOSTR
          BNE DECRDG
          CLR IOEXPD      Yes, clear exponent
          BRA DECRDG      Before finishing display

DECBIN    CLR IOSTR3      First clear output MSByte+1
          LDX #IOLSW      Set pointer to I/O work area
          STX TEMP2       Store pointer in temporary storage
          LDX #IOSTR      Set pointer to I/O storage
          LDAB #$04       Set precision counter
          JSR MOVEIT      Move I/O storage to work area
          LDX #IOSTR      Set pointer to original value
          LDAB #$04       Set precision counter
          JSR ROTATL      Start x10 routine (total =x2)
          LDX #IOSTR      Reset pointer
          LDAB #$04       And counter
          JSR ROTATL      Multiply by two again (total =x4)
          LDX #IOLSW      Set pointers
          STX TEMP2       For ADDER routine
          LDX #IOSTR
          LDAB #$04       Set precision counter
          JSR ADDER       Add original to rotated (total =x5)
          LDX #IOSTR      Reset pointer
          LDAB #$04       And counter
          JMP ROTATL      X2 once more (total =x10) and return

EXPOUT    LDAA #$C5       Set ASCII code for letter E
          JSR ECHO        Display E for exponent
          TST IOEXPD      Test if negative
          BMI EXOUTN      Yes, display '-' and negate
          LDAA #$AB       No, set ASCII code for '+'
          BRA AHEAD2      Go display it
EXOUTN    NEG IOEXPD      Negate to make exponent positive
          LDAA #$AD       Set ASCII code for '-'
AHEAD2    JSR ECHO        Display sign of exponent
          CLRB            Clear B to start counter
          LDAA IOEXPD     Get exponent

SUB12     SUBA #$0A       Subtract 10 (decimal)
          BMI TOMUCH      Look for negative result
          STAA IOEXPD     Restore positive result
          INCB            Advance 10's counter
          BRA SUB12       Keep subr to obtain MSDigit of exp

TOMUCH    TBA             Put MSDigit into A
          ADDA #$B0       Form ASCII code
          JSR ECHO        Output MS Digit
          LDAA IOEXPD     Get least significant digit
```

```
            ADDA  #$B0      Form ASCII code
            JMP ECHO        Output least signif. digit & return
```

This final routine ties the FPINP and FPOUT routines together, along with the floating point mathematical routines FPNORM, FPADD, FPSUB, FPMULT, and FPDIV to create an operating program that may be used as a floating point calculator program. All that is required by the reader is to supply the I/O driver routines, as previously described. The program will allow one to enter and receive data in the following format:

27.6E-2 X -5 = -0.1380000E+01

```
FPCONT  LDAA  #$8D      ASCII carriage return
        JSR ECHO        Output carriage return
        LDAA  #$8A      ASCII line feed
        JSR ECHO        Output line feed
        JSR FPINP       Get 1st floating point decimal number
        BSR SPACES      Display a few spaces
        LDX  #TPLSW     Set pointer to temporary storage
        STX TEMP2       Save in pointer storage area
        LDX  #FPLSW     Set pointer to FPACC LS Byte
        LDAB  #$04      Set precision counter
        JSR MOVEIT      Move FPACC to temporary storage
NVALID  JSR INPUT       Fetch operator from I/O
        CMPA  #$AB      Test for '+' sign
        BNE NOTADD      No, try '-'
        BSR OPERAT      Input FPACC value
        JSR FPADD       Add FPOP to FPACC
        BRA FINAL       Output result of addition

NOTADD  CMPA  #$AD      Test for '-' sign
        BNE NOTSUB      No, try 'X'
        BSR OPERAT      Input FPACC value
        JSR FPSUB       Subtract FPACC from FPOP
        BRA FINAL       Output result of subtraction

NOTSUB  CMPA  #$D8      Test for 'X' sign
        BNE NOTMUL      No, try '/'
        BSR OPERAT      Input FPACC value
        JSR FPMULT      Multiply FPOP times FPACC
        BSR FINAL       Output result of multiplication

NOTMUL  CMPA  #$AF      Test for '/' sign
        BNE NOTDIV      No, try delete
        BSR OPERAT      Input FPACC value
        JSR FPDIV       Divide FPOP by FPACC
```

```
FINAL   JSR FPOUT       Output the answer
        BRA FPCONT      Set up for new input

NOTDIV  CMPA #$8F       Not operator, try Control O
        BNE NVALID      No, ignore, try again
        BRA FPCONT      Yes, restart input string

OPERAT  JSR ECHO        Display control operator
        BSR SPACES      Display a few spaces
        JSR FPINP       Fetch second FP decimal number
        BSR SPACES      Display a few spaces
        LDAA #$BD       Set ASCII code for '='
        JSR ECHO        Display '=' sign
        BSR SPACES      Display a few spaces
        LDX #FOPLSW     Set pointer to FPOP LS Byte
        STX TEMP2       Store in temporary pointer storage
        LDX #TPLSW      Set pointer to first number input
        LDAB #$04       Set precision counter
        JMP MOVEIT      Move first input to FPOP and return
```

The three routines, FPINP, FPOUT, and FPCONT, as presented, require less than three pages of memory. This requirement may be shortened to some extent by forming subroutines for various common instruction sequences. This has not been done here to maintain clarity of operation. However, the ambitious reader should have little difficulty in shortening the program. The following list defines the data areas on page zero used by these routines. The addresses listed here are used by the relocatable floating point program presented in Appendix F.

ADDRESS	PROGRAM LABEL	DEFINITION
0010	INMTAS	I/O Mantissa Sign
0011	INEXPS	I/O Exponent Sign
0012	INPRDI	I/O Period Indicator
0013	IOLSW	I/O Work Area Least Significant Byte
0014	IONSW	I/O Work Area Next Significant Byte
0015	IOMSW	I/O Work Area Most Significant Byte
0016	IOEXP	I/O Work Area Exponent
0017	IOSTR	I/O Storage
0018	IOSTR1	I/O Storage
0019	IOSTR2	I/O Storage
001A	IOSTR3	I/O Storage
001B	IOEXPD	I/O Exponent Storage
001C	TPLSW	Temporary Input Storage Least Signif Byte

001D	TPNSW	Temporary Input Storage Next Signif Byte
001E	TPMSW	Temporary Input Storage Most Signif Byte
001F	TPEXP	Temporary Input Storage Exponent

This floating point program has been assembled as a relocatable program, and is presented in Appendix F as a memory dump. This means that the machine code for the operating portion of the program as presented in Appendix F may be loaded exactly as listed, without alteration, in any continuous block of memory. The only fixed memory requirement is the locations on page zero used to store the temporary data. This means that page zero must consist of RAM memory. The order in which the routines have been presented for explanation is not the same order in which they are assembled in Appendix F. However, the instruction sequence within each routine is exactly as listed in the text of this chapter, except for the BEQ PER1, JMP ENDINP instruction sequence following the label PERIOD. This is replaced with BNE ENDINP. Also, a complete symbol table is provided along with the memory dump.

DECIMAL ARITHMETIC ROUTINES

When using a computer to process mathematical data, such as data entered by an operator, and after processing, output for the operator to read, the decimal numbering system is most often the base used. This representation allows the operator to enter and read the data in a form most widely accepted and easily understood, since it is usually drummed into everyone from the time they are born. The computer, on the other hand, is generally designed to operate most efficiently with numbers in binary format. Therefore, there must be some means made available to allow the operator and the computer to communicate in a common number system.

As discussed in chapter four, conversion routines from one number base to another are often used. Routines, such as those presented, make it possible to input and output numbers in decimal notation while performing the actual calculations in binary notation. However, inaccuracies can creep into the most elementary calculation as a result of the conversion! For example, the subtraction of 2.1 from 5.0 may be output as 2.8999 rather than 2.9 due to conversion errors.

For applications where the operation required can be performed as decimal addition and subtraction, it would be far more accurate to perform these simple mathematical calculations in the same format as that used for input and output. The 6800 provides for this operation with an instruction that adjusts the contents of the accumulator to two binary coded decimal digits after an addition has been executed. This instruction also conditions the carry flag to provide for multiple precision decimal calculations. This instruction is the decimal adjust accumulator instruction, mnemonic DAA.

Presented in this chapter are routines that perform addition, subtraction, multiplication, and division of decimal numbers by utilizing the capability of the DAA instruction to operate with BCD digits. The format used to represent each number will be the same for all routines. Four bits are required to define each BCD digit. Therefore, two digits will be stored in a single eight-bit byte, with the least significant digit of the pair in the least significant half of the byte. These operations work with multiple precision values, allowing up to 256 bytes to be assigned for each number. For the routines presented, the bytes used to represent each number must

be stored in a table of sequential memory locations, with the byte containing the least significant digit pair in the lowest address of the table.

The first routine is the decimal addition routine. Using the DAA instruction, this routine adds the BCD digits contained in one table in memory to the BCD digits contained in another table. When calling this routine, the address of the least significant digit pair of one value must be stored in the temporary memory location labeled TEMP2, and the address of the least significant digit pair of the other value must be set up in the index register. The contents of this second value will be replaced by the result of the addition when the routine returns to the calling program. Also, the B accumulator must be set to the binary count of the number of bytes per table.

This routine first initializes the carry by clearing it before the addition of the first pair of digits. Each pair of digits indicated by the index register is then added to the corresponding digit pair indicated by TEMP2. The result of the addition of each pair of digits is adjusted to the proper decimal value by use of the DAA instruction before being stored. If the addition of the most significant digits results in the generation of an overflow, indicated by the carry flag being set, this condition of the carry flag will be maintained upon returning to the calling program. Since this condition may indicate an error, the calling program can check the C flag and take whatever action may be appropriate. The listing for this routine, which begins at the label DECADD, is presented next.

```
DECADD  CLC          Clear the carry flag
 DCAD1  LDAA X       Fetch byte of first number
        STX TEMP1    Store pointer to first number
        LDX TEMP2    Fetch pointer to second number
        ADCA X       Add byte from second number
        DAA          Decimal adjust the A accumulator
        STAA X       Store sum in second table
        INX          Advance pointer
        STX TEMP2    Store pointer to second number
        LDX TEMP1    Fetch pointer to first table
        INX          Advance pointer
        DECB         Decrement byte counter
        BNE DCAD1    Counter ≠0, continue addition
        RTS          =0, return
```

It may be noted that the decimal addition routine just presented is similar to the ADDER subroutine in chapter three. The only major difference between the two routines is the addition of the DAA instruction in this routine. However, the same cannot be said for the subtraction routine, SUBBER, and the decimal subtraction routine to be described next.

The decimal subtraction routine subtracts the subtrahend value in one table in memory from the minuend value contained in another table in memory. This subtraction is performed by converting the subtrahend to the complemented BCD value and adding the minuend to it. This method is similar to forming the two's complement of a binary number and adding it to the binary minuend, which effectively subtracts the original value of the two's complemented number from the other.

When the number is broken down into two digit pairs, as in these routines, the actual complement of each BCD digit pair can be in one of two forms. These two forms are referred to as the 100's and 99's complement of the BCD value. The 100's complement is used when the result of the subtraction of the previous pair of digits does not require a borrow from this pair. The 99's complement is used when the subtraction of the previous pair does require a borrow from this pair. For the 100's complement, the value being complemented is subtracted from the hexadecimal value 9A which is formed by adding one to the BCD value 99. The 99's complement of the number is formed by subtracting the number to be complemented from the BCD value of 99. The result of the subtraction of either form is used as the BCD complement of the subtrahend.

When the complemented value is added to the minuend, the result is the same as subtracting the initial subtrahend from the minuend. However, by adding the complement, the accumulator, carry flag, and half carry flag are conditioned for the proper execution of the DAA instruction.

After the DAA instruction is executed, the carry flag will indicate the borrow, or underflow, requirement of the next pair. If the carry flag is set to one, there is no borrow required, and the 100's complement must be formed for the next pair. If the carry flag is reset to zero, a borrow is required, and the 99's complement must be formed. It is important to note that for this operation the carry indicates the

opposite condition for the underflow, since addition of the complement is executed rather than subtraction of the actual subtrahend value.

The following calculation illustrates the process just discussed. The two four digit numbers are subtracted by forming the 100's or 99's complement, whichever is indicated, of the subtrahend, and adding it to the minuend. The least significant byte of the subtrahend is 100's complemented, since there is no borrow required. The second byte is 99's complemented, since the result of the subtraction of the first pair of digits requires a borrow from the second pair. The calculation presented is carried out two digits at a time.

4827 - 3246 = ?	
9A - 46 = 54	First, form 100's complement of the least significant pair of the subtrahend.
27 + 54 = 81	Minuend plus $\overline{\text{subtrahend}}$ = difference. No carry indicates next pair must be 99's complemented.
99 - 32 = 67	Form 99's complement of the next pair of the subtrahend.
48 + 67 = 115	Minuend + $\overline{\text{subtrahend}}$ = difference. The hundred's digit is dropped when result is formed. If an additional digit pair were to be subtracted, the presence of the hundred's digit here would mean the next subtrahend pair would be 100's complemented.
4827 - 3246 = 1581	Final result.

The subtraction routine listed next performs the decimal subtraction in the manner just described. When this routine is called, the address of the least significant digit pair of the minuend must be stored in the temporary storage area TEMP2; the index register must indicate the address of the least significant digit pair of the subtrahend; and the B accumulator must contain the binary count of the number of bytes for each number. The results of the subtraction are stored in the minuend table at the completion of the subroutine's operation. Before returning, the C flag will be set if a borrow is required for the subtraction of the most significant digit, or reset if the

borrow is not required, to inform the calling program of a possible error condition. The listing for the DECSUB subroutine is given next.

```
DECSUB  SEC          Set the carry flag
 DCSB1  LDAA #$99    Set BCD value of 99
        ADCA #$00    Add 00 to form 99 or 100 complemnt
        SBCA X       Of the subtrahend value
        STX TEMP1    Store pointer to subtrahend
        LDX TEMP2    Fetch pointer to minuend
        ADCA X       Add minuend to subtrahend
        DAA          Decimal adjust the difference
        STAA X       Store the difference
        INX          Advance the minuend pointer
        STX TEMP2    Store the minuend pointer
        LDX TEMP1    Fetch the subtrahend pointer
        INX          Advance the subtrahend pointer
        DECB         Decrement precision counter
        BNE DCSB1    ≠0, continue subtraction
        ROLA         Rotate the carry into A accumulator
        COMA         Complement the carry to condition
        RORA         Rotate the carry back to itself
        RTS          Return with result in subtrahend
```

The next pair of routines uses the decimal addition and subtraction routines to perform the actual computation. These routines add the capability to perform addition and subtraction of signed decimal numbers. The sign and magnitude of the numbers to be added or subtracted must be checked to determine whether the operation actually calls for an addition or subtraction, and to set up the proper sign for the result of the operation.

The two numbers to be operated on by these routines must be stored in two tables, referred to by the labels DCAC and DCOP. DCAC is the decimal accumulator, which is used to store one addend for the signed addition routine, and the minuend for the signed subtraction routine. DCOP is the decimal operand table, and must contain the other addend for the signed addition routine, and the minuend for the signed subtraction routine. For both routines, the results of the respective operations are stored in DCAC upon returning to the calling program. Also, the initial contents of DCOP are not necessarily maintained.

The number of bytes in each table can be varied to allow for the number of digits desired per value. For these routines, the tables

must be of equal length. The tables used by these routines are three bytes long, allowing six BCD digits per number. If the length of the tables is changed, the constant 03 in the instructions whose comments are marked by a double asterisk (**) must be changed to indicate the new byte count.

Unlike binary numbers, in which the MSB of the binary value may be considered as the sign bit, BCD representation does not allow for this convenient method of sign designation. One may sacrifice a BCD digit by assigning the MSB of the MS byte of a value as the sign bit. However, this method does not simplify the procedure for checking the sign of the value. It also complicates the process of checking for an overflow or underflow, since the C flag will not automatically indicate these errors. A separate memory location will be used to indicate the sign of the decimal values.

The sign of each number is set up in separate memory locations and uses the most significant bit of each byte to indicate a negative number if set, or a positive number if reset. The remaining bits in each sign byte must be all zeros, since there are several locations in the routine in which the sign bytes are checked as being equal, which involves the contents of the entire byte, not just the MSB. Also, making the remaining bits equal to zero is consistent with the format used by these routines to set and reset the sign bit of the result. The sign bytes that refer to the sign of the DCAC and DCOP are labeled SIGNAC and SIGNOP, respectively.

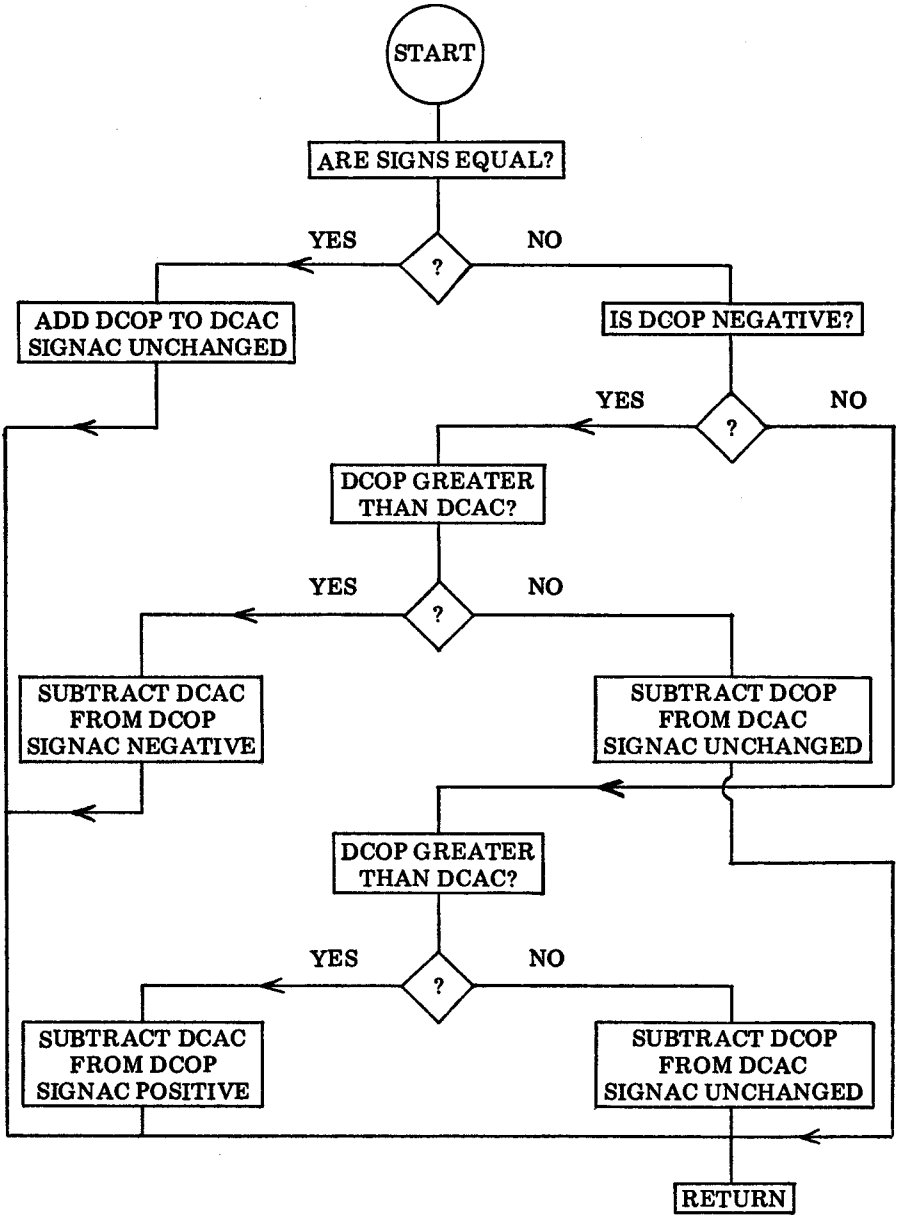
Depending on the sign and magnitude of the values operated on, it may be necessary to exchange the contents of DCAC and DCOP if the indicated operation is that of subtracting the accumulator from the operand. This exchange is accomplished by a subroutine labeled SHIFT. SHIFT exchanges the contents of the accumulator and operand one byte at a time.

In the process of determining which operation is actually called for (addition or subtraction), the relative magnitudes of the two numbers must be known. This is determined by the CMPR subroutine. Its operation is basically the same as that of the CPRMEM subroutine in chapter three. The only difference is that this routine is written specifically for comparing two triple precision values.

The signed addition routine, beginning at the label SGNADD, adds the contents of DCOP to DCAC, and returns with the answer in

DCAC. The calling routine simply loads DCAC, SIGNAC, DCOP, and SIGNOP with the desired values before calling this routine. When the sign of each is the same, the addition is performed as indicated. If the signs are different, the value of smaller magnitude is subtracted from the larger value, and the sign of the larger is set as the sign of the answer. The actual computation is done by one of the previous addition or subtraction subroutines. The condition of the carry flag upon returning to the calling program will indicate whether an overflow or underflow has occurred as a result of the operation, signalling a possible error condition. The operation of the signed addition routine is illustrated in the flow chart following the source listing.

```
SIGNOP   RMB $1          Sign byte of DCOP
SIGNAC   RMB $1          Sign byte of DCAC
DCOP     RMB $2          Decimal operand storage
DCOPM    RMB $1          Decimal operand MS Byte
DCAC     RMB $2          Decimal accumulator storage
DCACM    RMB $1          Decimal accumulator MS Byte

SGNADD   LDAA SIGNOP     Fetch sign of DCOP
         CMPA SIGNAC     Compare to sign of DCAC
         BEQ SAR2        Signs are equal, add numbers & return
         BCC SAR3        SIGNOP negative, SIGNAC positive
SAR1     BSR CMPR        Is DCOP greater than DCAC?
         BCS SB12        No, subtract DCOP from DCAC
         CLR SIGNAC      Yes, change SIGNAC to positive
SB21     BSR SHIFT       Exchange DCAC and DCOP contents

SB12     LDX #DCAC       Set pointers for subtracting
         STX TEMP2       DCOP from DCAC
         LDX #DCOP
         LDAB #$03   **  Set precision counter
         JMP DECSUB      Subtract and return

SAR2     LDX #DCAC       Set pointers for addition
         STX TEMP2       Of DCOP to DCAC
         LDX #DCOP
         LDAB #$03   **  Set precision counter
         JMP DECADD      Add and return

SAR3     BSR CMPR        Is DCOP greater than DCAC?
         BCS SB12        No, subtract DCOP from DCAC
         BEQ SB21        Equal, SIGNAC remains positive
         LDAA #$80       Yes, change SIGNAC
         STAA SIGNAC     To negative
         BRA SB21        Subtract DCAC from DCOP
```

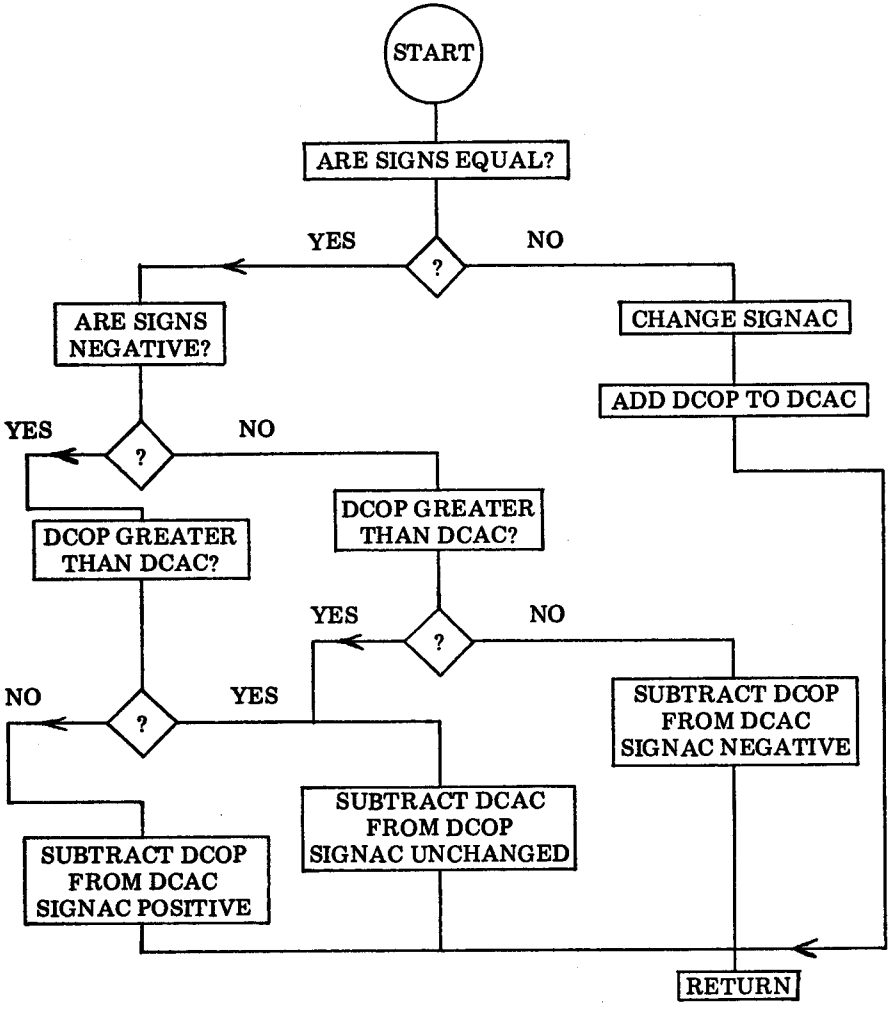

START
ARE SIGNS EQUAL?
YES
?
NO
ADD DCOP TO DCAC
SIGNAC UNCHANGED
IS DCOP NEGATIVE?
YES
?
NO
DCOP GREATER
THAN DCAC?
YES
?
NO
SUBTRACT DCAC
FROM DCOP
SIGNAC NEGATIVE
SUBTRACT DCOP
FROM DCAC
SIGNAC UNCHANGED
DCOP GREATER
THAN DCAC?
YES
?
NO
SUBTRACT DCAC
FROM DCOP
SIGNAC POSITIVE
SUBTRACT DCOP
FROM DCAC
SIGNAC UNCHANGED
RETURN

```
SHIFT   LDX #DCOP          Set pointer to DCOP
        LDAA X             Fetch byte from DCOP
        LDAB $03,X     **  Fetch byte from DCAC
        STAB X             Store DCAC byte in DCOP
        STAA $03,X     **  Store DCOP byte in DCAC
        INX                Advance pointer
        CPX #DCOPM+$1      Exchange complete?
        BNE SHIFT+$3       No, continue
        RTS                Yes, return

CMPR    LDX #DCOPM         Set pointer to MS Byte of DCOP
        LDAB #$03      **  Set precision counter
        LDAA X             Fetch byte from DCOP
        CMPA $03,X     **  Compare DCOP to DCAC
        BNE CMPRET         Not equal, ret w/ C flag conditioned
        DEX                Equal, decrement pointer
        DECB               Decrement precision counter
        BNE CMPR+$05       Last byte compared? No
CMPRET  RTS                Yes, return with C flag conditioned
```

The signed subtraction routine, starting at the label SGNSUB, subtracts the contents of DCOP from the contents of DCAC. The calling program must set the contents of DCAC, SIGNAC, DCOP, and SIGNOP with the desired values before calling this routine. The sign and magnitude of each of the numbers is examined to determine the actual operation to be performed. Several of the routines in the signed addition routine are used here. Since the decimal addition or subtraction routine is the last operation to be executed, the condition of the C flag will indicate whether an error has occurred. The listing and flow chart for the signed subtraction routine are presented next.

```
SGNSUB  LDAA SIGNOP        Fetch sign of DCOP
        CMPA SIGNAC        Compare to sign of DCAC
        BNE DIFSGN         Not equal, change SIGNAC and add
        TSTA               Are both negative?
        BMI NEGATV         Yes, compare magnitudes
        BSR CMPR           Both positive, is DCOP > DCAC?
        BCC SB21           Yes, subtract DCAC from DCOP
        LDAA #$80          No, set SIGNAC negative
        STAA SIGNAC
        BRA SB12           Subtract DCOP from DCAC

DIFSGN  LDAA SIGNAC        Fetch SIGNAC
        ADDA #$80          Change SIGNAC to opposite condition
        STAA SIGNAC        Store back in SIGNAC
        BRA SAR2           Add DCOP to DCAC
```

```
NEGATV   BSR CMPR      Compare DCAC to DCOP
         BEQ NEG1      Equal, make sign positive, result =0
         BCC SB21      DCOP > DCAC, sign negative,
                       Subtract DCAC from DCOP

NEG1     CLR SIGNAC    DCOP < DCAC, SIGNAC positive
         BRA SB12      Subtract DCOP from DCAC
```



Using these routines as a base, expanded decimal arithmetic programs can be written. One possible addition might be to include a decimal point by specifying either a fixed number of digits in the DCAC and DCOP to be to the right or left of the decimal point, or setting up a memory location to define the exponent. The exponent may reside in one or more bytes of memory and also have a sign byte associated with it. By following the procedures outlined in Chapter 5, one may develop a floating point program using decimal values for the mantissa and exponent. The following routines may be used to perform the multiplication and division operations of this type of floating point program.

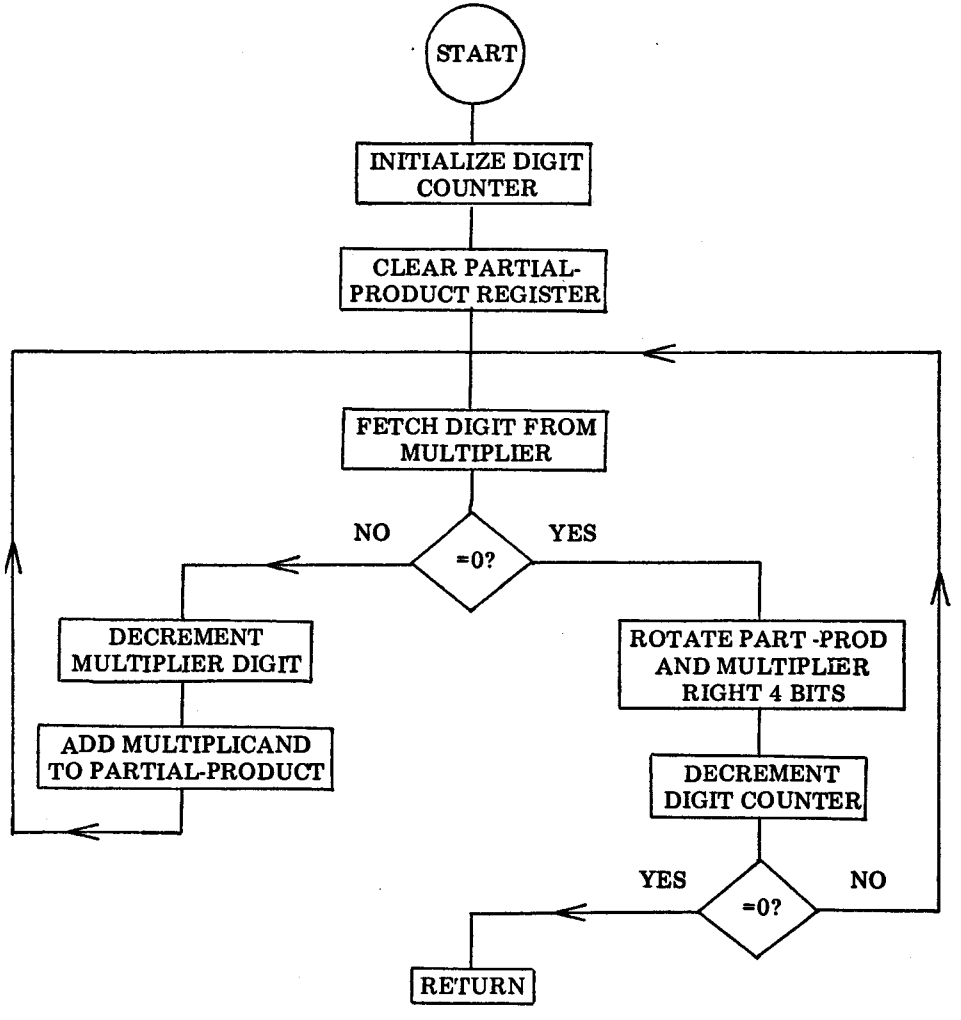
The multiplication and division routines both operate with a four byte accumulator and operand. The first three bytes contain the six BCD digits of the respective values. The fourth byte is an extension of each value to allow for an overflow during the calculations. The fourth byte must be cleared before entering either of these routines. Also, a memory location is set aside for both routines to store a digit counter value. This location, labeled DIGCNT, is initially set by these routines to the number of significant digits of the accumulator. As the operations proceed, this value is decremented; when it reaches zero, the operation is complete.

A table area labeled DCPP is used to store the partial-product and quotient for the respective operations. This table consists of seven bytes, which allows enough room for the multiplication routine to maintain any overflow that may occur.

The multiplication routine multiplies the contents of the decimal operand by the contents of the decimal accumulator. Beginning with the least significant digit of DCAC and working up, each digit is used as a counter for the number of times the operand is to be added to the partial-product. When the counter goes to zero, the contents of the partial-product are rotated right, which achieves the same result as multiplying the operand by ten. The next digit of the accumulator is then selected as a counter for the number of times the operand is added to the partial-product. This multiplication loop is executed once for each significant digit of the decimal accumulator. At the completion, the contents of DCPP0 to DCPP5 contain the 12 significant digits of the result. If this routine is used in part of a floating point program, the results may be normalized by shifting the partial-product register to the left until a non-zero BCD digit is in the most significant half of the most significant byte. This normalization pro-

cess follows the same general outline as that defined in Chapter 5. The listing and flow chart for the decimal multiplication routine are presented next.

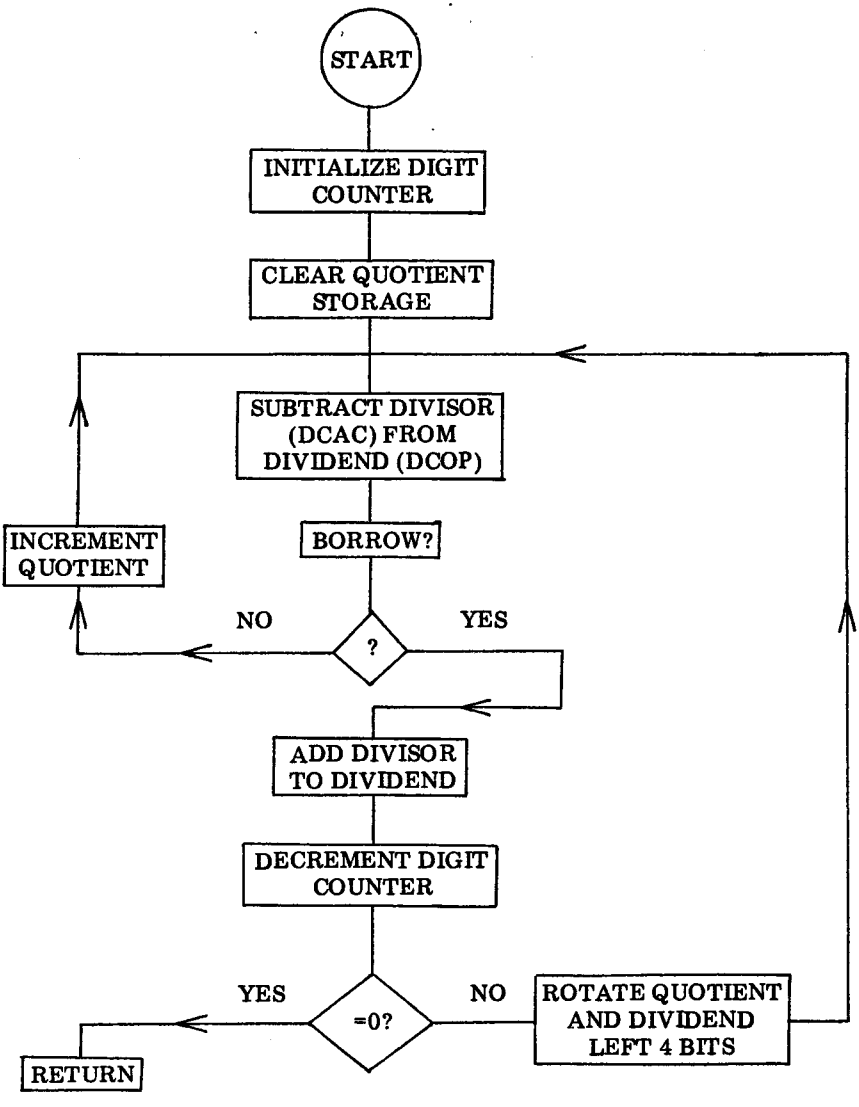
```
DIGCNT  RMB $1          Digit counter
TMPCNT  RMB $1          Temporary counter storage
DCPP0   RMB $1          Partial product LS Byte
DCPP1   RMB $1
DCPP2   RMB $1
DCPP3   RMB $1
DCPP4   RMB $1
DCPP5   RMB $1          Partial product MS Byte
DCPP6   RMB $1          Partial product extension
DCOP    RMB $2          DCOP storage
DCOPM   RMB $2          DCOP MS Byte and extension
DCAC    RMB $2          DCAC storage
DCACM   RMB $2          DCAC MS Byte and extension

DECMUL  LDAB #$6        Set digit counter
        STAB DIGCNT     Store in memory
        INCB            Set precision counter
        LDX #DCPP0      Set pointer to partial-product
        JSR CLRMEM      Clear partial-product area
NXTDGT  LDAB DCAC       Fetch LS Byte of accumulator
        ANDB #$0F       Mask off upper half
        BEQ DIGDON      If 0, no need to multiply this digit
        STAB TMPCNT     Store digit in temporary counter
MULTPL  LDX #DCPP3      Set pointer to operand
        STX TEMP2       Store in temporary storage
        LDX #DCOP       Set pointer to partial-product storage
        LDAB #$4        Set precision counter
        BSR DECADD      Add DCOP to partial-product
        DEC TMPCNT      Decrement digit multiplier
        BNE MULTPL      ≠0, continue multiply loop
DIGDON  LDAA #$4        Set rotate counter
PPSHFT  LDAB #$7        Set precision counter
        LDX #DCPP6      Set pointer to partial-product
        JSR ROTATR      Rotate partial-product right
        LDAB #$3        Set precision counter and
        LDX #DCACM      Pointer to DCAC and
        JSR ROTATR      Shift the accumulator to the right
        DECA            Shifted 4 times?
        BNE PPSHFT      No, continue rotating right
        DEC DIGCNT      Decrement digit counter
        BNE NXTDGT      ≠0, continue multiplication
        RTS             Return
```



The decimal division routine operates in a manner similar to the binary to decimal conversion routine of Chapter 4. That is, it subtracts the divisor from the dividend until a borrow is required. A count of the number of times the subtraction is successfully performed is maintained, and becomes part of the quotient. When the borrow is detected, the routine rotates the dividend four bits to the left, and the subtraction cycle begins again. As each digit of the quotient is generated, it is shifted into the least significant digit of the quotient.

Before calling this routine, the divisor and dividend must be loaded into the DCAC and DCOP as normalized decimal numbers. Once again, to normalize these decimal values, the most significant non-zero BCD digit must be in the most significant digit location of the respective values. At the completion of this routine, the quotient is contained in DCPP1 through DCPP3. As compensation for the operation of the routine, a value of one must be added to the exponent of the quotient. The listing and flow chart for this decimal division routine are presented next.

```
DECDIV   LDAB  #$06      Set up digit counter
         STAB  DIGCNT    Store digit counter in memory
         LDX   #DCPP0    Set pointer to QUOTIENT
         JSR   CLRMEM    Clear QUOTIENT storage

DVNEXT   LDX   #DCOP     Set pointer to DIVIDEND
         STX   TEMP2     Store in memory
         LDX   #DCAC     Set pointer to DIVISOR
         LDAB  #$04      Set precision counter
         BSR   DECSUB    Subtract DIVISOR from DIVIDEND
         BCS   SUBDON    If borrow, exit subtraction
         INC   DCPP1     Increment decimal counter
         BRA   DVNEXT    Continue divide loop

SUBDON   LDX   #DCOP     Set precision counter
         STX   TEMP2     Store pointer in memory
         LDX   #DCAC     Set pointer to DIVIDEND
         LDAB  #$04      Set precision counter
         JSR   DECADD    Add DIVISOR back to DIVIDEND

         DEC   DIGCNT    Decrement digit counter
         BEQ   DVEXIT    =0, return
         LDAA  #$04      Set rotate right counter
RESULT   LDX   #DCPP1    Set pointer to QUOTIENT
         LDAB  #$03      Set precision counter
         JSR   ROTATL    Rotate QUOTIENT right
         LDX   #DCOP     Set pointer to DIVIDEND
         LDAB  #$04      Set precision counter
         JSR   ROTATL    Rotate DIVIDEND left
         DECA            Decrement rotate counter
         BNE   RESULT    ≠0, continue rotating QUOTIENT
         BRA   DVNEXT    Continue division loop

DVEXIT   RTS             Return
```


START
INITIALIZE DIGIT COUNTER
CLEAR QUOTIENT STORAGE
SUBTRACT DIVISOR (DCAC) FROM DIVIDEND (DCOP)
BORROW?
INCREMENT QUOTIENT
NO
?
YES
ADD DIVISOR TO DIVIDEND
DECREMENT DIGIT COUNTER
YES
=0?
NO
ROTATE QUOTIENT AND DIVIDEND LEFT 4 BITS
RETURN

INPUT/OUTPUT PROCESSING

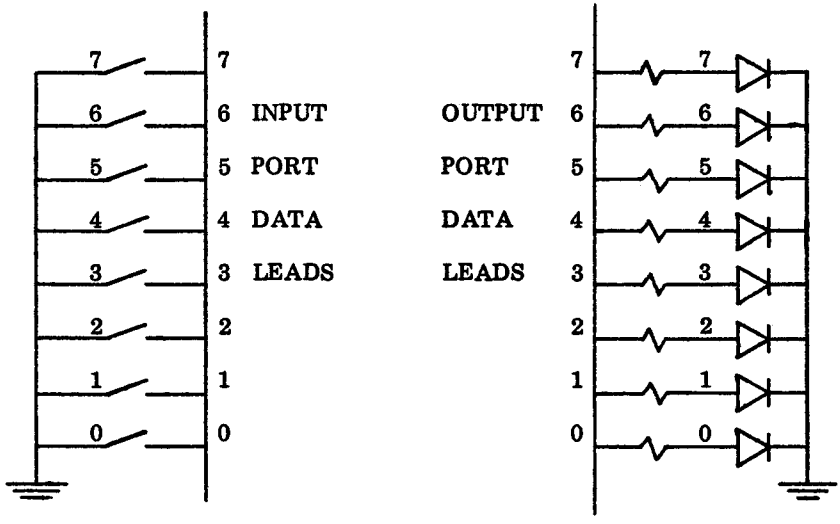
Writing a program to communicate with a peripheral device is as important as almost any other programming task one may have to perform. Nearly every program one can think of requires some form of input or output. The input data may be received from a group of sensors that make up a burglar alarm system, or it may be entered through a keyboard device for a variety of control or data entry purposes, or from a bulk storage device, such as magnetic tape, for loading programs or reading large blocks of data. The output data may be used to turn relays or lights on and off, to send characters to a display device (such as a mechanical printer or video display), or to store programs or data on a bulk storage device. No matter what the task, it is important to be able to write effective I/O driver programs.

Before the various forms of I/O routines are presented, it is important to understand the input/output setup of the 6800. The 6800 CPU handles input and output in the same manner as reading and writing to memory. This means that any addressable memory location may be used as an eight-bit parallel I/O port. Therefore, it is possible to have 64K of eight-bit parallel I/O devices on one system, although this would be highly impractical since some memory would be required to store the program to operate the I/O devices. This method of accessing an I/O port as though it is a location in memory allows the use of any of the memory access instructions to transfer data to and from the I/O devices. With this capability, the programmer is afforded considerable flexibility in testing and transferring data with an I/O device.

For purposes of this discussion, the following convention will be assumed for the I/O ports. An input port consists of eight parallel data lines that provide TRUE LOGIC to the 6800 CPU. TRUE LOGIC menas that a logic '1' transmitted by the input device is seen by the 6800 as a logic '1.' An output port is assumed to consist of eight parallel data lines that receive data written to it by the 6800 and maintain the eight bit data at the output port lines until another data pattern is written to the output port.

The first type of I/O processing to be discussed is one that would be used in conjunction with the simplest form of input and output devices. The input device might be a group of switches, or sensors, that provide a '1' or '0' to each of the input data lines to indicate an

open or closed position. The output device might consist of a group of lamps that may be turned on by outputting a '1,' or off by outputting a '0.' The schematic below illustrates this configuration.



As indicated by this schematic, the input port has eight switches, numbered 0 through 7, connected to its corresponding eight data leads. These switches might be sensor switches in a burglar alarm system that monitor the opening and closing of doors throughout a building. Assuming that the switches are closed when the doors are properly secured, the following input routine may be used to test for an open door. The label SWINP refers to the memory address of the switch input port.

```
SWTEST  LDAA SWINP      Read switch input port
        BEQ SWTEST      If 0, all doors closed, continue testing
        . . .           One or more doors open, alarm cndtn
```

This routine illustrates the simplicity of inputting information from an input port. The data is read into the A accumulator by the LDAA instruction. Each bit of the A accumulator now indicates the open (1) or closed (0) condition of the switches connected to the input port, and the status flags are conditioned to indicate whether one of the switches is open. For this example, the Z flag will be set to '1' if all the switches are closed. Should any of the switches become

open, the data lead corresponding to that switch will go to a '1' condition, and the Z flag will be reset since the A accumulator will not be 0.

This instruction sequence assumes that the data read from the input port must be loaded into the A accumulator for further examination if an alarm condition is detected. If this is not necessary, the TST instruction may be substituted. This would perform the same test without altering the contents of the A accumulator.

One should be aware that it is not necessary to use all eight data leads of an input port. Suppose there are only five switches, zero through four, connected to the input port, and the other three leads are not used. In this case, a different test procedure would be required. The program listing below loads the A accumulator with a value of 1F, and the BITA instruction is used to test for a one in any of the five least significant bits of the input port. The Z flag would, again, indicate the possible open condition of one or more of the five switches.

```
SWTEST  LDAA #$1F       Set the bit test byte
        BITA SWINP      Test 5 least significant bits
        BEQ SWTEST+$2   If 0, all doors closed, continue testing
        . . .           One or more doors open, alarm cndtn
```

If only one data lead was required, by connecting it to bit seven of the input port, the N flag could be used in testing for a '1' or '0' condition. In this case, the conditional branch instruction in the first listing would be changed to a BPL instruction.

Over on the output port side of the schematic, a set of eight lights, shown here as light emitting diodes, is connected to the eight output port data leads, numbered zero through seven. Each light is turned on by outputting a '1' to the corresponding data lead. The light is turned off by outputting a '0.' For example, to turn on the odd numbered lights, and turn off the even numbered lights, one could load the B accumulator with a bit pattern of '1 0 1 0 1 0 1 0' and store it in the output port, as listed below. The label LIGHTS refers to the memory location assigned to the output port.

```
        . . .
        LDAB #$AA       Load the desired bit pattern
        STAB LIGHTS     Output pattern to lights
        . . .
```

If these lights are connected to the control panel of the burglar alarm system described previously, they could be used to indicate which of the doors have been opened by including an instruction to output the data to the lights as it is read from the switches. The following sequence may be used.

```
SWTEST   LDAA SWINP     Read switch input port
         STAA LIGHTS    Output switch conditions to display
         BEQ SWTEST     If 0, all doors closed, loop back
         ...            One or more doors open, alarm cndtn
```

After inputting the data from the switches, the routine immediately outputs the same data to the lights. In so doing, any light that turns on will indicate that the corresponding door is open. The program then tests for a door open, just as before, and either continues testing, if the doors are all closed, or performs whatever logic may be necessary when a door is found to be open (i.e. sounding an alarm, calling the police, etc.).

Naturally, the switches and lights used in this example may be replaced by a wide variety of devices for an even greater number of applications. For instance, the input may come from heat, light, or pressure transducers, or from such devices as analog-to-digital converters, which transform an analog signal to a proportional digital binary, or BCD, value. An output port may drive relays, 7 segment displays, alarms, or digital-to-analog converters.

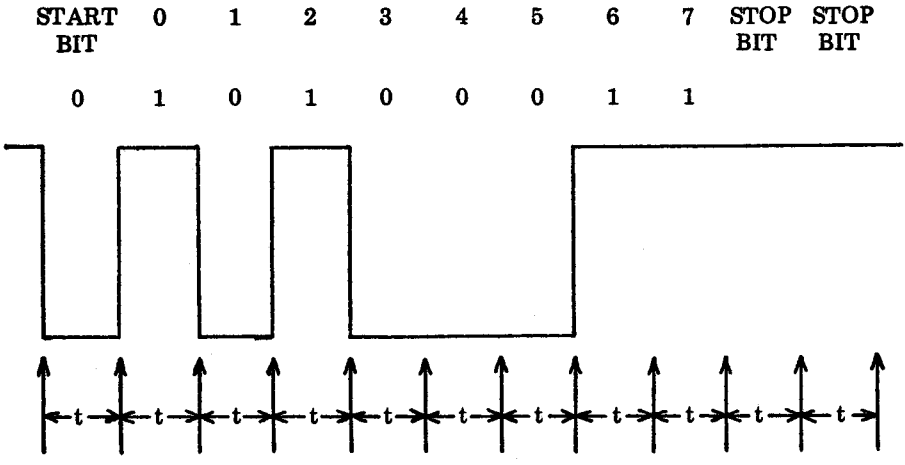
A very novel application for a simple output device is to connect a speaker to one bit of an output port and have the computer synthesize different frequencies to create music. The different tones are generated by outputting alternate ones and zeros with an appropriate delay (as described in chapter three) in between each output. The shorter the delay, the higher the frequency, and vice versa. By outputting a given set of tones in the proper sequence with each tone lasting the proper duration, a musical tune can be played by the computer. Many 6800 based microcomputers have been known to play such intriguing songs as "Mary had a little lamb" and "A bicycle built for two." Looking at this application from a more scientific viewpoint, this form of frequency synthesis may be used to generate any number of different waveforms for a multitude of technical applications.

One such technical application is in the generation of asynchronous serial data. Serial data is data that is sent one bit at a time with each bit lasting a specific amount of time before the next bit is output. Asynchronous serial data is a short group of bits output in serial form. Each group of bits generally represents a single character of one of the standard character sets (i.e. ASCII, BAUDOT) although random data patterns may be transmitted in this fashion. The reason it is referred to as asynchronous is because the beginning of the group of bits may occur at any time, although once started, the timing of each bit in the group must meet the specified time. The timing diagram shown next illustrates the manner in which the ASCII code for the letter 'E' (11000101 in binary) is transmitted as asynchronous serial data.

As noted in the timing diagram, the character code for the 'E' is preceeded by a start bit. This bit is used to inform the receiving device that a character is being transmitted. The character code then follows the start bit, beginning with the least significant bit. The character transmission is completed by adding one or more stop bits to the end of the code. The stop bits are added to allow time for the receiving device to prepare to receive another character.

The timing diagram also indicates that there is a specific amount of time, 't,' for the duration of each bit. This timing is often referred to by the number of bits that could be transmitted in one second at this rate, rather than the amount of time used for each bit. The standard bit per second, or BAUD, rates used for transmitting ASCII code range from 110 BAUD for many keyboard and printer devices, to 9600 BAUD for high speed devices.

The computer may be used to generate serial data in this form, by outputting one bit at a time, and providing a programmed delay between each bit to create the proper timing. The routine listed next outputs eight bit characters as asynchronous serial data with two stop bits. The timing generated by this routine outputs data at a rate of 110 bits per second. This corresponds to a delay between bits of 9.09 milliseconds. The timing may be calculated by adding up the number of cycles per instructions (indicated in the column of figures to the left of the listing) for each instruction executed between the output of each bit. This timing assumes a cycle time of one microsecond.



This routine may be used to output ASCII characters to a printer or other type of device that receives asynchronous serial data at 110 bits per second. The character to be output must be in the A accumulator when this routine is called. The initial contents of the B accumulator are pushed onto the stack at the start of this routine, and then pulled from the stack before returning. The output of each bit is accomplished by rotating it into bit zero of the A accumulator, and storing it in the memory location assigned to the output port. One should make special note of the fact that the instructions between the output and rotate operations do not affect the carry flag. This allows the routine to maintain the character in the A accumulator upon returning to the calling program.

```
                PRINT   PSHB            Save initial contents of B
                        CLC             Clear carry for start bit
                        ROLA            Rotate carry into A
                        BSR BITOUT      Output start bit
2                       LDAB  #$08      Set data bit counter
8               PRINT1  BSR BITOUT      Output data bit and delay
2                       DECB            Decrement bit counter
4                       BNE PRINT1      ≠ 0, output next bit
2                       LDAB  #$01      Set up stop bit in B
4                       STAB PRINTR     Output stop bit
2                       RORA            Reposition character in A
8                       BSR TIMER       Delay for first stop bit
8                       BSR TIMER       Delay for second stop bit
                        PULB            Restore initial contents of B
5               DUMMY   RTS             Return to calling program
```

```
4   BITOUT  STAA PRINTR     Output bit to printer
2           RORA            Position for next output
8           BSR TIMER       Delay one bit time
5           RTS             Return

4   TIMER   PSHB            Save bit counter
2           LDAB #$C9       Set delay counter value
8   TIME1   BSR DUMMY       Branch to return instruction to
8           BSR DUMMY       Provide delay using
8           BSR DUMMY       Accumulator B for delay counter
2           DECB            Decrement delay counter
4           BNE TIME1       ≠ 0, continue delay loop
4           PULB            Done, restore bit counter
5           RTS             Return
```

The type of peripheral devices discussed up to this point require nothing more than a simple input or output instruction to transfer the information. When a transfer is to be made, the program does not care what state the peripheral is in previous to the transfer. However, for many peripherals, the process of transferring data between it and a computer under program control requires some type of hand-shaking. This means that a program must check whether the device is ready to make a data transfer, and, when so indicated, perform the logic necessary to make the transfer. In general, there are two methods used to provide the program control. One method is to have the program continuously input the status bit of the peripheral, often referred to as the "programmed data transfer" (or PDT) bit, until it indicates the device is ready for a data transfer. The other method is for the peripheral device to send a signal to the computer when it is ready for a data transfer. This signal is called an interrupt. Once an interrupt is received, the method of data transfer is similar to that for the PDT operation. As will be indicated, the major difference between the two modes is that under PDT operation the program must continuously check the status of the device, while under interrupt operation the program is free to perform other operations while waiting for the interrupt from the peripheral.

Whether a peripheral device is designed to generate interrupts or operate strictly in the PDT mode, there is generally a PDT bit associated with it. A device that generates interrupts will have a PDT bit to provide the option of operating in the PDT mode and, when operating under interrupt, to identify itself as the device that generated the interrupt, should there be more than one interrupting device in the system. It is, therefore, important to understand how to

check the PDT bit of a device. Any peripheral that is designed to operate with a PDT bit will have a status output. This output may contain only the PDT bit, or it may include several other status leads to indicate error conditions that may occur in the peripheral. These status leads are connected to an input port allowing the status to be examined by a program. There are several ways of checking the PDT bit, depending on its location within the memory byte. If the PDT is located in the most significant bit of the status byte, by loading an accumulator with the status, the N flag will indicate the condition of the PDT bit.

```
CKPDT   LDAA STATUS     Load status byte into A
        BPL CKPDT       If PDT =0, continue testing it
        ...             PDT =1, device ready, begin processing
```

If the PDT bit is located in a bit position other than bit 7, the BIT test instruction may be used. One of the accumulators must be loaded with all zeros except for the bit corresponding to the location of the PDT bit in the device's status byte. Then, by performing the BIT test between the accumulator and the device's status, the Z flag will indicate the opposite condition of the PDT bit. The following routine checks the PDT bit at bit 1 until it indicates that the device is ready.

```
CKPDT   LDAA #$02       Set bit 1 to test the PDT
        BITA STATUS     Condition the Z flag for the PDT test
        BEQ CKPDT       If Z set, device not ready
        ...             If Z reset, device is ready
```

There are times when it is known that a PDT bit must change within a certain amount of time. For instance, after outputting a character to a display device there is usually a specific maximum time limit for the device to accept it and the PDT bit to come true again. If this time limit is surpassed, it might indicate a problem with the display device. This possible error may be monitored by the program by inserting a counter in the PDT test loop. The counter would be calculated to allow only a given amount of time to elapse before the PDT bit must return, otherwise an error routine would be

entered to inform the operator of a possible problem. The following format may be used to include a timer in the PDT checking routine. The exact timing of this loop may be calculated as discussed in chapter three.

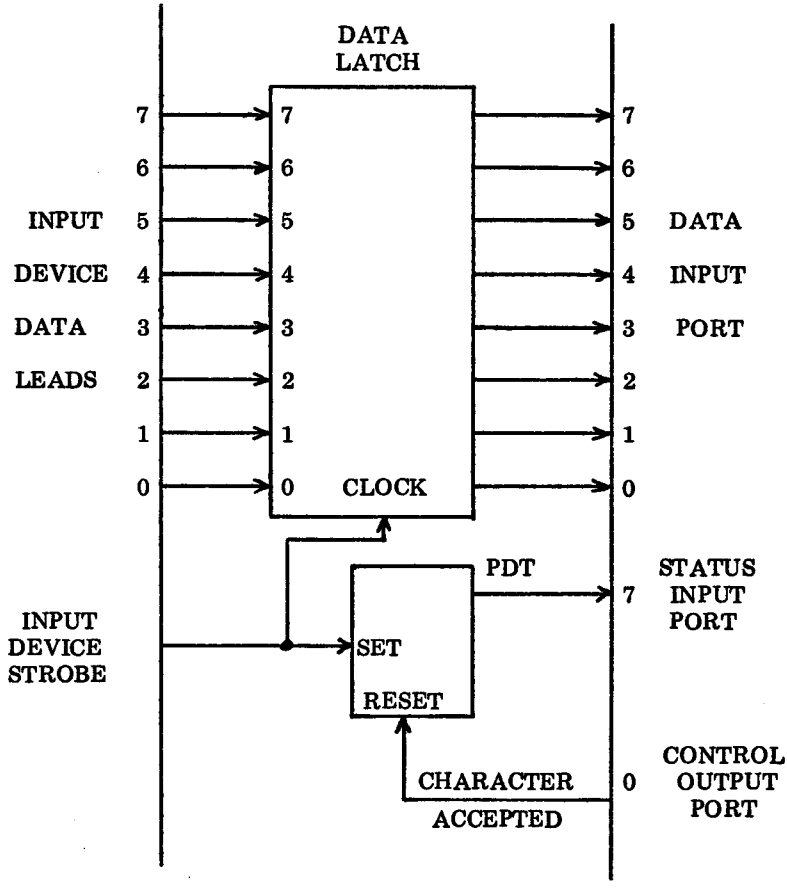
```
LPSET   LDAB #$YY       Set up timing loop counter
CKPDT   LDAA STATUS     Condition N flag for PDT test
        BMI PDTSET      Have PDT, continue processing
        DECB            No PDT, decrement timer
        BNE CKPDT       Timer ≠ 0, continue testing
        . . .           Time out, possible error
```

PDT operation, for most input devices, generally follows the same basic procedure. When a program requires data from an input device, it reads the status of the device and checks the condition of the PDT bit. If the PDT bit indicates the device has data available, the program can proceed to input the data. For some devices, once the data has been read in, a "character accepted" signal from the program may be required to reset the PDT bit.

This procedure is typical of many interfaces that latch the data in from a device and then set a PDT bit. The schematic shown next illustrates this form of interface. The data is entered into the latches by the input device by setting up the data at the input to the latches and then pulsing the strobe line of the latches. This same strobe signal sets the PDT bit. After the data has been read by the program, the reset line is pulsed by the program outputting a "character accepted" signal.

A program to control this type of interface is listed next. The PDT bit is connected to bit 7 of the status input port. This allows the program to check for the PDT bit by reading the status and testing for the condition of the N flag. The data is entered through the data input port. Once the data has been accepted, the PDT bit is reset by outputting a "character accepted" signal to the control output port.

```
PDTINP  LDAA STATUS     Input device status
        BPL PDTINP      N = 0, no PDT, continue testing
        LDAA DATAIN     N = 1, read data from device
RESET   LDAB #$01       Set up output pulse
        STAB CHRACC     Output character accepted
        CLR CHRACC      Reset character accepted to create pulse
        RET
```

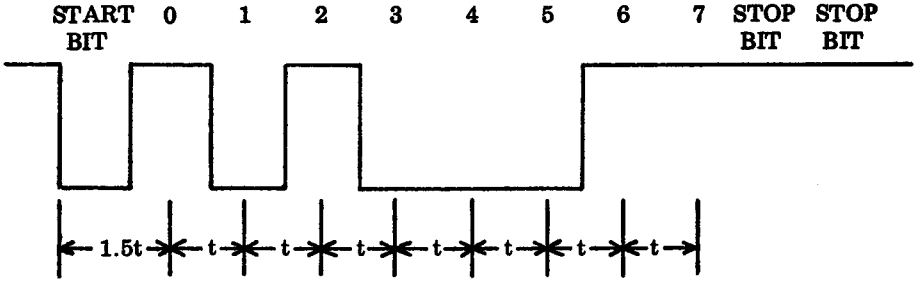


In this program listing, the "character accepted" signal is derived by loading the B accumulator with 01, and outputting it to the control output port, referred to by the label RESET, then executing a CLEAR instruction which returns bit 0 of the control output port to 0. This effectively creates a pulse on the least significant data lead of the control output port to provide a "character accepted" signal. Some interfaces, however, are reset by simply writing to the control output port. In this case, all that would be required to send a "character accepted" signal would be to execute the CLEAR instruction after reading the data. A third possibility is that the interface resets the PDT bit when the input is executed. For this setup, the input routine may be exited just after the data is read.

The label RESET has been included in this routine to point out the portion of the routine that resets the PDT bit. This portion may be required as an initial reset for the input device at the start of a program that uses the device to receive data. Quite often when dealing with such devices, it is necessary to output a reset during the initializing stages of the program. This guarantees that the device status will indicate the true status when the device is first called upon to input some data. For the other cases described above, in which the PDT is reset by writing to the control output or reading from the data input, the corresponding instruction should be executed to initialize the input device.

Another type of routine that may be considered to test the status before inputting the data is a program that inputs asynchronous serial data. The start bit of the asynchronous data could be considered its PDT bit. The input routine would test for the presence of the start bit and, when received, the data bits that follow may be read in by sampling the data at the proper time intervals.

Sampling the data is performed by providing a programmed delay until the mid-point of each bit is reached, and then inputting the value of the bit as it appears on the input data lead. The arrows in the timing diagram shown next indicate when each bit should be sampled by the program. The left most arrow indicates when the start bit is first detected. The next arrow indicates a delay time equal to one and a half bits before sampling bit zero of the data. Each subsequent sample is taken after a delay time of one bit.



The program listed next may be used to receive asynchronous serial data, eight bits at a time, such as that generated by the PRINT routine, previously presented in this chapter. The timing provided in this routine reads the data at 110 BAUD. By altering the delay, as described in chapter three, this timing may be changed to input data over a wide range of BAUD rates. The number of cycles for each instruction is indicated in the left-hand column. The delay time between sampling may be calculated by adding up the number of cycles for each instruction executed between inputs. The major portion of the delay is provided by the same TIMER subroutine used in the PRINT routine.

The data is input through bit seven of the data input port. This allows the program to simply test the N flag for the start of the bit. Each bit is then inputted by rotating bit seven of the input port into the carry and then rotating the carry into the A accumulator. When the last bit has been inputted, an additional delay of one bit time is added to make sure the input data is into the stop bit before returning to the calling program. If this final delay was not provided, and the last data bit of the input was a '0,' the calling program could call SRLINP again while the data bit is still at the input port. If this occurs, SRLINP would input the last data bit and assume it to be the start bit of a new character, which would result in the input of erroneous data. The data received is returned to the calling program in the A accumulator.

```
     SRLINP  PSHB          Save initial value of B
4            LDAA KYBDIN    Input to look for start bit
4            BMI SRLINP+$1  N flag = 1, no start bit yet
2            CLRA           Have start bit, clear A
8            BSR HAFBIT     Delay ½ bit time
8            BSR TIMER      Delay 1 bit time
2            LDAB #$08      Set data bit counter
6    NEXBIT  ROL KYBDIN     Move data bit into carry
2            RORA           Move data bit from carry into A
8            BSR TIMER      Delay one bit time
2            DECB           Decrement bit counter
4            BNE NEXBIT     ≠ 0, input next bit
             BSR TIMER      Delay into stop bit
             PULB           Restore initial value of B
             RTS            Return with character in A

4    HAFBIT  PSHB           Save bit counter
2            LDAB #$64      Set half bit time delay
4            BRA TIME1      Delay one half bit time
```

PDT operation of a parallel output device is generally straightforward. When the PDT bit is checked and indicates the device is ready to accept data, the data may be output. Upon receipt of the data by the device, the PDT bit will change state to indicate that the device is busy processing the data. Once the processing is completed, the PDT will return to its device ready status and wait for the next output from the program. Therefore, if the program is to output more than one character, the PDT bit must be monitored after each character is output to determine when the device is ready to accept the next character.

The following routine might be used to output a line of text to a printer that accepts ASCII characters as eight bit parallel data. This routine fetches the characters one at a time from a buffer and outputs them to the printer. When a carriage return is detected in the character string, it is output, followed by a line feed, and then the program returns to the calling program. The PDT bit is assumed to be in bit 0 of the status input from the printer. Also, when the routine is called, the index register is assumed to be pointing to the start of the character string.

```
LINOUT   BSR CKPDT      Wait for printer PDT
         LDAA X         Fetch character from message storage
         CMPA #$8D      Character = carriage return?
         BEQ FINISH     Yes, complete output
         STAA CHROUT    No, output character to printer
         INX            Advance character string pointer
         BRA LINOUT     Wait for PDT

FINISH   STAA CHROUT    Output carriage return
         BSR CKPDT      Check PDT before sending line feed
         LDAA #$8A      Set line feed character
         STAA CHROUT    Output line feed
         RTS            Return to calling program

CKPDT    LDAA #$01      Set up to test PDT
         BITA STATUS    Test status of printer PDT
         BEQ CKPDT+$1   PDT = 0, wait for printer
         RTS            PDT = 1, return to output next char
```

This method of checking the PDT bit of an I/O device to transfer data is commonly used when it is not required to perform other functions while waiting for a data transfer from a peripheral. When

there is no background program to be executed while waiting for a peripheral, it is of no consequence for the program to remain in a loop for long periods of time, testing the PDT bit of a peripheral. In order for the data to be transferred as rapidly as possible, the program must maintain constant attention to the PDT bit. This is also necessary if the data is only available for a given length of time, as is the case when using a card reader to enter data. The program must read a character from the card reader when it is available. Once the reader starts reading a card, it does not stop in between each character while the program reads it. The program must be ready for each character when it is available, otherwise, the character will be lost.

Another method of transferring data under program control is to have the I/O device send an interrupt signal to the computer when it is ready for a data transfer. This signal interrupts the program currently in progress and directs the CPU to an interrupt service routine. The interrupt routine performs the logic necessary to transfer the data to or from the peripheral and then returns to the original program as though it had not been interrupted at all. This method of operation is known as interrupt processing.

Interrupt processing is analogous to a postman being interrupted while delivering the mail. As the postman is placing the mail in a row of mailboxes, someone walks up to him and taps him on the shoulder. The mailman completes filling the current mailbox, makes a mental note as to which mailbox is to be filled next, and turns to the person. The mailman is given a letter and is asked to send it. The mailman takes the letter and stores it in his mailbag. He then returns to the job of filling the mailboxes, beginning with the box he remembers as the next one to be filled. This is similar to the procedure followed by a computer when an interrupt is received from a peripheral.

When an interrupt signal is received by a computer, the current instruction being executed is completed and the address of the next instruction to be executed is saved. It is also necessary to save the information contained in the CPU registers and status flags. This information must be saved so that it may be properly restored before returning to the interrupted program. The computer is now ready to perform the steps necessary to transfer the data between itself and the peripheral. Once the transfer is completed, the CPU registers and status flags must be restored to their initial contents at the time the interrupt was received. Execution of the interrupted program is resumed at the instruction that would have been executed next, if

the interrupt had not been received.

Before presenting methods of interrupt processing with the 6800, several features of the 6800 designed for this mode of operation should be discussed. These features make interrupt processing with the 6800 easy and effective.

There are two types of hardware interrupts available on the 6800. One is the non-maskable interrupt. When received, the non-maskable interrupt is always acknowledged by the 6800. For this reason, the non-maskable interrupt is generally used for very high speed devices that require immediate attention, or as a power failure interrupt to allow the storage of any critical information when a power failure is detected. The other hardware interrupt is the maskable interrupt. Its acknowledgement is dependent on the condition of the I flag. When the I flag is set, the maskable interrupt line is disabled and an interrupt on the maskable interrupt line will not be acknowledged by the 6800. When the I flag is reset, the 6800 will acknowledge a maskable interrupt. The maskable interrupt is generally used by most of the devices that operate under interrupt control. One should note that whether a device uses a maskable or non-maskable interrupt, the interrupt service routine for either will be basically the same.

The condition of the Interrupt flag may be software controlled by one of three instructions. The SEI instruction sets the I flag, disabling the maskable interrupt. The CLI instruction clears the I flag, enabling the receipt of maskable interrupts. The third instruction, namely TAP, conditions not only the I flag, but all of the status flags as well, by loading the condition code register with the contents of the A accumulator.

When writing a program to operate with interrupts, there are several times when it may not be desired to accept interrupts. One is during the initialization of the program, before all the necessary pointers, counters, and tables used by the program have been set up. If an interrupt is received before the program is ready to accept it, the program may receive or transmit erroneous data. To avoid such an occurance, the first instruction of the program should be the disable interrupt instruction. Then, after the initialization is complete, the interrupts may be enabled since the program is now ready to deal with the interrupts properly.

Another time that interrupts must be disabled is upon receipt of

an interrupt. This is to allow the program enough time to respond to the first interrupt before receiving a second. The 6800 automatically disables the maskable interrupt upon receipt of the maskable, non-maskable, and software interrupts. Therefore, it is not necessary to include a SEI instruction in the interrupt service routine. When the interrupt service routine is finished, and is ready to return to the interrupted program, the return from interrupt instruction will restore the I flag to its initial condition at the time the interrupt was received. If it is desired to allow "nesting" of an interrupt, the interrupt service routine can enable the maskable interrupt after it has completed its initial steps and can, itself, be interrupted. The process of nesting interrupts will be discussed later in this chapter.

It may also be necessary to disable interrupts when a section of the program is changing information vital to the function of the interrupt routine. This information might be the address for storing or retrieving data to be transferred, or a flag indicating the progress of the program to the interrupt routine. For whatever reason, the program must disable interrupts before the change is made, make the change to the information, and, upon completion, enable the interrupts. This will provide the smooth transition of information needed by the interrupt routine.

Another feature of the 6800 is the automatic register storage that takes place when an interrupt is acknowledged. The contents of the program counter, index register, A accumulator, B accumulator, and condition code register are pushed onto the stack when an interrupt is acknowledged. It is necessary to save this information so that it may be restored upon returning to the interrupted program. This automatic storage relieves the interrupt service routine of the burden of storing the CPU registers and status flags. This function may be executed prior to the receipt of an interrupt by a WAIt for interrupt instruction. This instruction stores the CPU registers and halts the operation of the 6800 until an interrupt is received. When the interrupt is received, the 6800 will go directly to the interrupt service routine since the CPU registers have already been stored.

When the interrupt service routine has completed its operation and is ready to return to the interrupted program, the CPU registers and status flags must be restored to their initial conditions at the time of the interrupt. This is accomplished by the use of the special return from interrupt instruction, mnemonic RTI. This instruction pulls the information from the stack, loads it into the

proper registers, and returns to the interrupted program at the instruction following the last instruction executed.

The procedure for receiving interrupts from a 6800 based microcomputer follows the basic steps outlined above. When an interrupt is received from a peripheral, the CPU automatically pushes the contents of the program counter, stack pointer, A accumulator, B accumulator, and condition code register onto the stack, and sets the I flag. (For the maskable interrupt, this procedure assumes that the I flag is reset at the time the maskable interrupt occurs.) The CPU then vectors to the proper interrupt service routine.

The interrupt service routine performs the logic required to service the interrupting device. It is usually a combination of the PDT routine for the device being controlled, and a routine that checks and stores the data for an input, or sets up the data to be output. Since the interrupt routine is meant to operate independently from the main program, it must perform its own checks and manipulate the data into and out of memory, as well as driving the peripheral. In order to accomplish this, and to provide a flexible interrupt routine, a link between the main program and the operation of the interrupt service routine must be established.

One method of establishing the link is through the use of an interrupt table area. This table area normally includes at least three items, namely, a memory pointer, a data counter, and an in-progress flag. The memory pointer is used by the interrupt routine to indicate where input data is to be stored, or where output data is to be found. As the interrupt routine stores or outputs each byte of data, the memory pointer is advanced to the next location. The data counter indicates to the interrupt routine the amount of data to be received or sent. The routine decrements this counter each time it inputs or outputs some data. When the counter reaches zero, the operation is complete. If necessary, the end of the operation may also be indicated by the receipt or transmission of a terminating character, such as a carriage return or line feed, which would terminate the operation before the data counter reached zero. The completion of the operation is then signalled by resetting the in-progress flag. The in-progress flag is set by the main program when the input or output is initiated. Then, when the interrupt routine is finished with the I/O operation, the in-progress flag is reset. The main program periodically checks this in-progress flag and when it is reset, the main program knows that the I/O operation is complete.

The in-progress flag may also serve another purpose. The interrupt routine can test this flag when an interrupt is received to determine whether an interrupt from the peripheral is expected. If it is expected, the interrupt routine can service the interrupt normally. If the interrupt is not expected, the interrupt may be ignored by resetting the I/O device, if necessary, and returning to the interrupted program, or by entering an error routine, which informs either the main program or the computer operator of the erroneous interrupt.

After the interrupt service routine completes its operation, it returns control to the interrupted program. This is accomplished by executing the return from interrupt instruction, which pulls the original status and CPU register values from the stack, and returns to the interrupted program. Restoration of the status and CPU registers before exiting the interrupt service routine allows the interrupted program to continue execution as though the interrupt never occurred. Having defined the basic procedures for dealing with an interrupt, a more detailed look at the programming for an input or output device will now be presented.

Interrupt processing for an input device is not exactly the same as that for an output device. The reason for this difference is that an interrupt from an input device indicates that the input device has a character or some data available for the program. The program may read the data in, process it, and then wait for another interrupt. For an output device, an interrupt indicates that the device has accepted the previous output and is ready to receive another character. Therefore, an output device must initially receive an output from the program before it generates an interrupt. Also, after the last character is received by the output device, a final interrupt will be generated, which must be ignored. This difference is further illustrated by the following input and output interrupt routines.

The input interrupt service routine presented next stores characters, as they are inputted, in a buffer area in memory until either the buffer is filled or a carriage return is received. This routine might be used to input characters from a keyboard, or data from a paper tape reader. This routine uses a table area which contains the input buffer pointer, data counter, and in-progress flag. This table is listed next, followed by the table setup routine of the main program. The table setup routine initializes the contents of the table when an input sequence is to begin. Several facts about this routine and the table contents should be noted.

First, the in-progress flag in the first byte of the table is represented by the sign bit, not the contents of the entire byte. Therefore, the remaining seven bits in this byte may be used to signal error conditions or intermediate progress status, if this type of information is required by either the interrupt routine or main program.

Next, the input buffer pointer is stored in the second and third bytes of the table, with the page portion of the address in the second byte, and the low portion in the third byte. The address that must be initially loaded into these locations is the start address of the input buffer minus one. Setting this pointer to the location before the start of the input buffer is necessary because the input interrupt routine increments the input buffer pointer before storing the character received, not after.

Finally, one should note the use of the set and clear the I flag instructions in the SETINT routine before and after the table is set up. This prevents an interrupt from being acknowledged while the contents of the input interrupt table are being initialized. However, the necessity of disabling interrupts while setting up this table may be eliminated by setting up the data counter first, and working backwards. In this way, the in-progress flag will not be set until the other pertinent data has been loaded.

INTERRUPT INPUT TABLE

```
FLAGIN   RMB $1          In-progress flag, sign bit
         RMB $1          Page portion, input buffer pointer
         RMB $1          Low portion, input buffer pointer
         RMB $1          Data counter

         ...             Set up routine for input
SETINT   SEI             Disable mskbl interrupts during setup
         LDAA #$80       Byte to set in-progress flag
         STAA FLAGIN     Store in-progress flag
         LDX  #INPBFR-$1 Set input buffer -1 pointer
         STX FLAGIN+$1   Store pointer in table
         LDAA #$XX       Set data counter
         STAA FLAGIN+$3  Store counter in table
         CLI             Enable maskable interrupts
         ...             Continue main program
```

The input interrupt service routine is listed shortly, followed by

the flow chart. In this listing, the input is performed by a single load instruction. This assumes that the input device is reset by reading the data from its input port. When implementing this routine, the instruction marked by the double asterisk (**) should be replaced by those required to operate the specific input device being driven.

It is assumed in this routine that only one device in the system can generate an interrupt. Therefore, it is not necessary to check for the PDT bit of the input device. If one desires to check the PDT bit, as an error checking measure, this routine should include an instruction sequence before the actual data is inputted, which inputs the PDT bit of the input device and tests the status. If the PDT bit is not set properly, an error routine should be entered. Otherwise, the routine should proceed to input the data and continue with the normal interrupt processing.

After the data has been inputted, the in-progress flag is checked to determine whether an input is expected by the interrupt program. This routine ignores an unexpected interrupt by simply returning to the interrupted program without storing the character inputted. It should be noted that by performing the input sequence before checking the in-progress flag, the input device will be properly reset whether the interrupt was expected or not.

Assuming the interrupt was expected, the character received is stored in the input buffer. The new input buffer pointer is then stored in the input interrupt table. The data counter is decremented and, if zero, the in-progress flag is reset and the interrupt service routine is exited. If it is not zero, the character just received is tested for a terminating character. In this routine, the input may be terminated by a carriage return, ASCII code 8D. If it is a carriage return, the in-progress flag is reset to end the input operation and the interrupt routine is exited. If it is not a carriage return, the in-progress flag remains set when the routine is exited.

The short instruction sequence following the interrupt service routine listing may be used by the main program to check for the completion of the input operation. When the sign bit of the in-progress byte is reset, the main program will branch to the appropriate routine, referred to here as CMPTIN, to examine the data received. The contents of the interrupt input table may be used by the main program in examining the data input, since the input buffer pointer indicates the location of the last character received, and the data counter

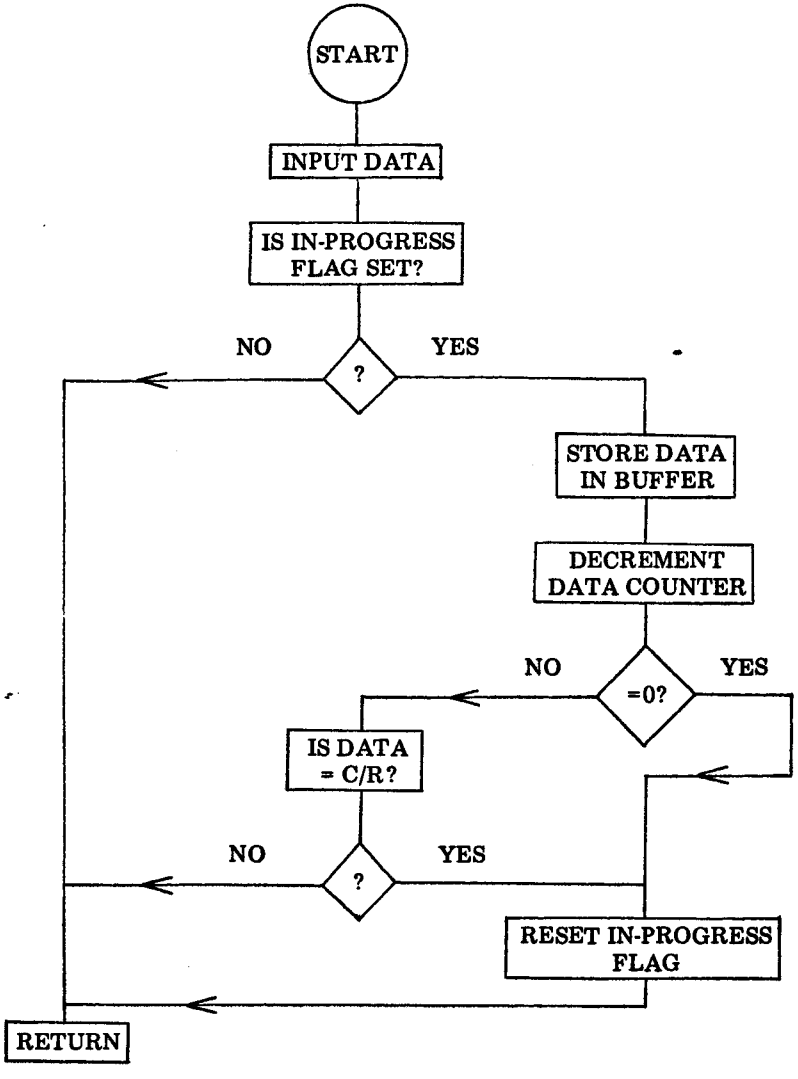
indicates either the number of unused locations in the input buffer, or, if equal to zero, that the entire buffer is filled.

```
INTINP   LDAA INPDAT   ** Input data from input device
         TST FLAGIN       Check in-progress flag
         BPL EXITIN       Interrupt not expected, ignore
         LDX FLAGIN+$1    Set pointer to input buffer
         INX              Advance input buffer pointer
         STX FLAGIN+$1    Save input buffer pointer
         DEC FLAGIN+$3    Decrement character counter
         BEQ FININP       If zero, input finished
         CMPA #$8D        Is character a carriage return?
         BNE EXITIN       No, exit input routine
FININP   CLR FLAGIN       Input complete, clear in-progress flag
EXITIN   RTI              Return to interrupted program

         ...              Sequence to check
         ...              In-progress flag for end of operation
         TST FLAGIN       Check sign bit of in-progress byte
         BPL CMPTIN       Sign reset, input complete
         ...              Sign set, continue other processing
```

Operation of an output device under interrupt control requires a different sequence of events from that for an input device. As pointed out before, the main reason for this difference is that the output device generates an interrupt after a character has been outputted by the program, while the input device generates an interrupt to indicate that it has a character available. The output routine presented next illustrates the different approach that must be taken for an output device.

This output interrupt routine outputs a string of characters stored in a buffer in memory. This routine may be used to output messages to a printer or video display, or to output data to a low or medium speed storage device. (High speed devices generally use a method of "direct memory access" in which the data is transferred directly from memory to the storage device, or vice versa, under control of a hardware interface.)

The interrupt output table is the same type of table used to provide the exchange of information between the main program and the input interrupt service routine. The organization of the table is the same as the input table, with the in-progress flag, output buffer



pointer, and data counter. However, when the table is initialized, the buffer pointer is set to the actual start address of the output buffer, rather than the start address minus one as in the input table.

Aside from setting up the table, the initialization routine checks the in-progress flag to determine whether an output is currently being executed. This may occur when a program uses the same out-

put device to display messages from a number of different routines, such as error and advisory messages in a system monitor program. This check eliminates the possibility that an output will be initiated before a previous one is finished. This operation has not been included in the input routine since it is less likely that two separate inputs will be required at the same time. However, if the possibility does exist, a similar instruction sequence should be added to the input initialization routine before the disable interrupt instruction.

When the in-progress flag is reset, the output may be initiated. First, the output table is set up with the required information. While this table is being loaded it is not necessary to disable interrupts since the output device should not generate an interrupt until after the first character has been sent. Once the proper information is contained in the table, the first character is actually output by this routine. This output is performed by the STAA OUTDAT instruction in this listing. This output starts the output sequence which is then carried on by the interrupt service routine. For implementation of this routine on one's own system, the instructions in this routine and in the interrupt service routine marked with the ** should be changed to the instruction sequence necessary to drive the specific output device used.

OUTPUT INTERRUPT TABLE

```
FLGOUT   RMB $1              Output in-progress flag
         RMB $1              Page portion, output buffer pointer
         RMB $1              Low portion, output buffer pointer
         RMB $1              Output data counter

         ...                 Output initialization routine
TSTOUT   TST FLGOUT          Check in-progress flag
         BMI TSTOUT          If output in progress, wait
         LDAA #$XX           Set character counter
         STAA FLGOUT+$3      Store in output interrupt table
         LDX #OUTBFR         Set output buffer pointer
         STX FLGOUT+$1       Store pointer in output interrupt table
         LDAA #$80           Set in-progress flag
         STAA FLGOUT         Store in output interrupt table
         LDAA X              Fetch first character to output
         STAA OUTDAT  **     Output character to device
         ...                 Continue main program
```

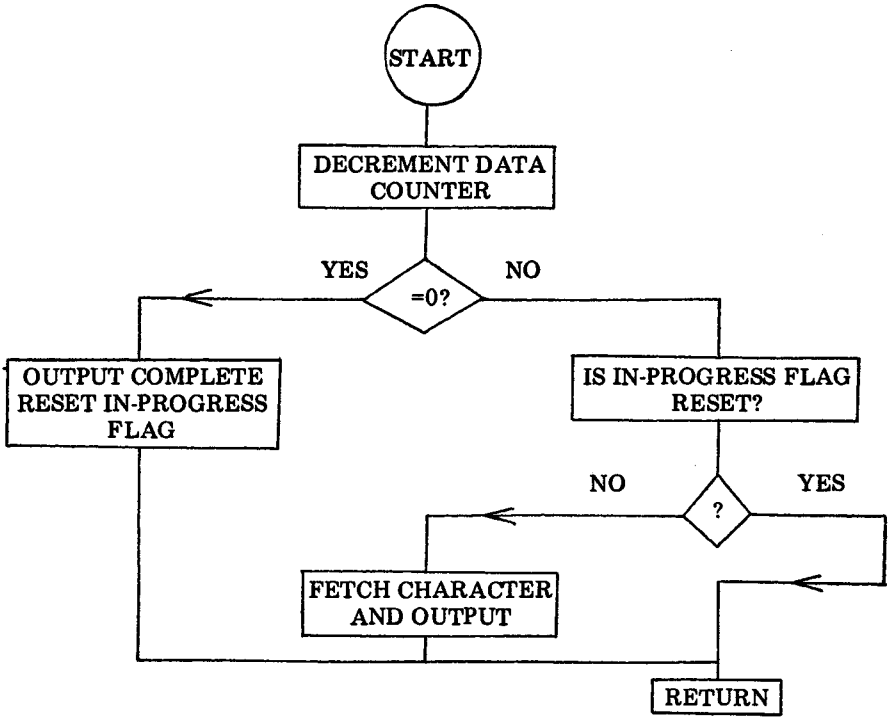
The output interrupt service routine is entered upon receipt of an interrupt from the output device. The data counter is decremented once and checked for zero. When it reaches zero, the last character has been outputted, and the output operation is complete. The in-progress flag is reset, and the routine returns to the interrupted program.

If the counter is not zero, the in-progress flag is checked to make sure that the output routine is expecting an interrupt. As in the input interrupt service routine, this is indicated by the in-progress flag being set. If it is reset, the interrupt may be ignored by simply returning to the interrupted program, or an error routine may be entered that signals either the main program or the operator that an unexpected interrupt was received.

If the routine makes it by the test, the next character may be outputted. In this routine, it is assumed that the output device is the only device generating interrupts. Thus, a PDT test is not necessary before outputting the character. However, if it is felt that such a test should be performed before outputting the character, the required instruction sequence for testing the PDT bit may be included. The routine then returns to the interrupted program. The listing and flow chart for this output interrupt routine are presented next.

```
INTOUT   DEC FLGOUT+$3     Decrement character counter
         BEQ FLGRST        =0? Yes, reset in-progress and exit
         LDX FLGOUT+$1     Fetch output buffer pointer
         TST FLGOUT        Check in-progress flag
         BPL EXITOT        Reset, ignore interrupt
         INX               Advance output buffer pointer
         LDAA X            Fetch character to be output
         STX FLGOUT+$1     Save output buffer pointer
         STAA OUTDAT  **   Output character
EXITOT   RTI               Return to interrupted program

FLGRST   CLR FLGOUT        Reset in-progress flag
         RTI               Return to interrupted program
```

Up to this point, only one device has been considered to generate an interrupt. When an interrupt is received, the interrupt service routine simply performs the indicated input or output for the single device. This may not always be the case, since an I/O controller quite



often controls an input and an output device, and generates an interrupt for both devices. Or, there may be several interrupting devices connected to the system. In order to operate more than one type, the interrupt service routine must determine which device generated the interrupt by "polling" each device when an interrupt is received.

Polling means that the interrupt service routine checks the status of each device that could have generated the interrupt. This is done by checking the PDT bit of each of the devices. When a PDT bit is found to be set, the appropriate service routine is entered to execute the I/O for that device. At the conclusion of the service routine, the return from interrupt instruction is executed to return to the interrupted program.

The following listing is an example of a polling routine that checks the status of three possible interrupting devices. This instruction sequence should be the initial sequence of the interrupt routine. The labels DVICE1, DVICE2, and DVICE3 refer to the interrupt service

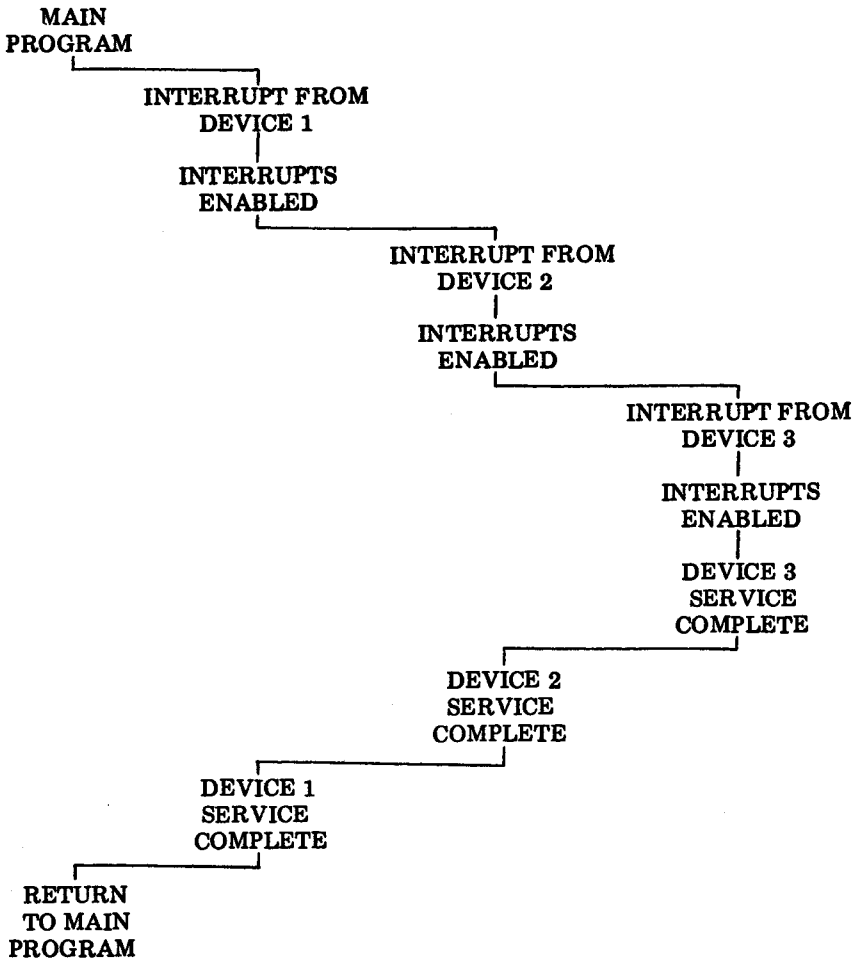
routines that perform the I/O logic for the designated device. This routine tests the PDT bit of each device, assumed to be in the sign bit of the device status, and jumps to the proper service routine when a PDT bit is set. If none of the possible devices have the PDT bit set, this routine ignores the interrupt and returns to the interrupted program. This condition may be treated as an error condition, if necessary, rather than ignoring it.

```
...             Polling routine
TST PDTDV1      Test status of device 1
BMI DVICE1      If PDT set, service device 1
TST PDTDV2      Test status of device 2
BMI DVICE2      If PDT set, service device 2
TST PDTDV3      Test status of device 3
BMI DVICE3      If PDT set, service device 3
RTI             None set, ignore interrupt
```

The use of several interrupting devices in a system may require the capability of the interrupt service routines to allow receipt of an interrupt from one device while another device is currently being serviced. This means that the service routine of the first interrupting device must enable interrupts before it has completed its operation. Then, if an interrupt from a second device occurs before this routine is finished, the current interrupt routine being executed becomes the interrupted program of the second interrupt. Allowing interrupts to overlap in this manner is referred to as NESTING interrupts.

The illustration below shows how the flow from one interrupt routine to another would proceed if three interrupting devices generated interrupts within a short period of time, to create the nesting of three levels of interrupts.

The 6800 stack plays an important role in nesting interrupts. Saving the CPU registers and status in the stack as each interrupt is received allows the interrupt routines to interrupt each other without setting up special pointers and data storage areas for each interrupt level or device. The only restrictions on the number of nesting levels as far as the stack is concerned, is the amount of memory provided for use by the stack. Each interrupt uses seven memory locations in the stack to store the registers. Therefore, for every interrupt nesting level one can expect, there must be seven memory locations available in the stack, plus that required for the other uses of the stack by the



main program (i.e. subroutine calls, temporary data storage).

Deciding when to enable interrupts in an interrupt service routine to allow nesting is generally determined by the speed of the device being serviced. A low speed device may allow interrupts to be enabled immediately upon entering the interrupt service routine, since it is likely that the data transfer is not required immediately. A medium speed device, or a device that has a limited amount of time to transfer the data, may require the data transfer to be performed before the interrupts are enabled. Then, the remainder of the service routine may be executed while the interrupts are enabled. If a high

speed device is being operated in the interrupt mode, it is very likely that it should not allow interrupts to be enabled until the end of its service routine. If it were to enable interrupts, a slower device might delay the execution of the high speed device's service routine to the point that a second interrupt from the high speed device would be received before the initial interrupt had been serviced completely. Therefore, one should carefully consider which routines, and where in the routines, the interrupts are to be enabled.

This method of selecting when to enable interrupts is a means of setting a priority for the interrupting devices. The high speed devices would have the highest priority, since they do not allow themselves to be interrupted until the service routine is finished. The medium speed devices, which may enable interrupts after several operations of the service routine have been completed, would be considered a middle priority. The low speed devices would be the lowest priority because they may be interrupted at any time during the interrupt service routine.

This system of priorities may be augmented by the computer's hardware if a priority interrupt interface is used, which fields the interrupts from the interrupting devices and allows the higher priority interrupts through first, before those of lower priority. This type of interface greatly eases the burden on the interrupt software of setting up priorities for the interrupts received.

When deciding whether to operate a computer system's peripherals under PDT control or interrupt, one should consider the type of programs (along with the number of peripherals) to be used in the system. If the programs are the type that receive an input and then output a response to a single terminal, the PDT mode would be the easiest to implement, and would provide sufficient performance. Programs of this type include games, editors, and small system monitors. For programs that provide keyboard entry and storage or retrieval from a bulk storage device to enter and store mailing lists, for example, one should consider interrupt processing. This would allow the data entry and bulk storage to be performed simultaneously. However, if the speed of the storage device is fast enough to store each entry with a minimal delay imposed between entries, the PDT mode may work just as well. For programs that operate a number of peripherals simultaneously and, in essence, are running more than one or two programs at a time, the interrupt mode of operation is a

necessity. Such "multi-programmed" systems might be used to control several terminals at once, while monitoring a burglar or fire alarm system. Therefore, one should carefully consider the overall requirements for the type of programs to be run when setting up the I/O portions of one's system.

SEARCH AND SORT ROUTINES

The capability of a computer to manipulate data stored in its memory is another reason why the computer is such a powerful machine. The speed with which it can search large blocks of data and extract information, or sort the data into alphabetical order, or into common groupings, is far beyond the capability of a human. The information may represent a wide variety of data. For example, the data may be lists of names and addresses that are to be searched for specific geographic regions or sorted into alphabetical order. Or, the data may be numerical information, such as test grades or data gathered for a research project, which is to be analyzed to obtain the average or derive other information about the data through statistical analysis. In order for the computer to perform these tasks, the information to be processed must be arranged in memory in a specific format, and programs must be written to perform the desired operation.

The data to be manipulated must be arranged in some form of table in memory. The table may contain a number of entries. Each entry may consist of one or more bytes of memory, depending on the maximum size of a single entry and the format specified for an entry. The two types of tables to be discussed in this chapter are commonly referred to as FIXED FORMAT and FREE FORMAT tables. In a fixed format table, the data is arranged in a standard fashion for each entry. The same number of bytes is assigned to each entry, no matter how many bytes an entry may actually take up. A free format table allows the size of each table entry to follow the data pattern of the entry. If the first entry requires four bytes and the next requires six, in a free format table, the first entry will only use the four bytes and the second will use six. There are advantages to both formats, depending on the application. These will be discussed as the search routines for each format are presented.

In order to provide a means of comparing the two formats, two search routines will be presented that perform the same function. However, one routine utilizes a search table of fixed format and the other utilizes a free format search table. The function performed by these search routines is that of receiving a command input from a

keyboard and searching a control table for the same command. If the command is found, the start address of the command routine is taken from the table and is used to jump to that command routine. If the command is not found in the table, the search routine returns to wait for another command input. These routines have many practical applications since, as will be seen, the entries in the control table may be easily modified to represent as many commands as one may need for a specific program. Any program that allows an operator to input commands to direct its operation may find one of these routines useful.

The following control tables shall be used to illustrate the operation of the fixed format and free format search routines. The commands in these tables are: GO, LIST, MEDIAN, AVG, COUNT, and CLEAR. These commands might be used to direct the computer to aid in conducting an experiment. The GO command could initiate a ten second sampling interval, during which time a sensor is monitored to detect the occurrence of an event. A count of the number of times the event occurs within the ten second interval is stored in the computer by the GO command routine. The LIST routine might be used to print out the counts stored for each ten second interval up to that time, allowing one to examine the raw data for possible patterns that may develop. The MEDIAN and AVG commands could calculate the median and average values of the counts stored for each interval, and output the value to a printer. The COUNT command might be used to indicate the number of ten second intervals that have been initiated up to that time. The CLEAR command could be used to reset the storage area to allow a new set of tests to begin.

In both the fixed format and free format control tables, each entry is divided into two FIELDS. The first field consists of the character string that defines the command name. In the fixed format entry, this field is set to a fixed length. In this case, it is six characters long. For the command names that do not use all six locations available for the name, the unused locations are filled with zeros. In the free format entry, the command field contains the characters for the name plus one more location that is filled with zeros. This extra location is used to indicate the end of the name, as will be discussed shortly. The second field is the same for both formats. This field is two bytes long and contains the start address of the command

in the entry.

The end of each control table is indicated by a zero byte stored immediately following the last entry. By terminating the tables in this way, the search routines may simply check the first character of each entry for a zero byte to determine when the end of the table is reached. Therefore, the number of entries in the control table is completely independent of the operation of the search routine. Modifying the number of commands in the control table is accomplished by adding or deleting the command entries and moving the zero byte to the end of the new control table. The search routine does not have to be changed at all.

The control table for the fixed format search routine is presented next, followed by the control table for the free format routine. One should make note of the differences in length of the two tables, due to the extra zeros that must be added to the fixed format entries. This and other points to be considered when deciding which format to use will be discussed following these tables.

FIXED FORMAT CONTROL TABLE

0200	C7	Code for letter G
0201	CF	Code for letter O
0202	00	Not used for this command
0203	00	Not used for this command
0204	00	Not used for this command
0205	00	Not used for this command
0206	01	Page where GO routine starts
0207	40	Location on page where GO starts
0208	CC	Code for letter L
0209	C9	Code for letter I
020A	D3	Code for letter S
020B	D4	Code for letter T
020C	00	Not used for this command
020D	00	Not used for this command
020E	01	Page where LIST routine starts
020F	60	Location on page where LIST starts
0210	CD	Code for letter M
0211	C5	Code for letter E
0212	C4	Code for letter D
0213	C9	Code for letter I
0214	C1	Code for letter A
0215	CE	Code for letter N
0216	01	Page where MEDIAN routine starts

0217	80	Location on page where MEDIAN starts
0218	C1	Code for letter A
0219	D6	Code for letter V
021A	C7	Code for letter G
021B	00	Not used for this command
021C	00	Not used for this command
021D	00	Not used for this command
021E	01	Page where AVG routine starts
021F	A0	Location on page where AVG starts
0220	C3	Code for letter C
0221	CF	Code for letter O
0222	D5	Code for letter U
0223	CE	Code for letter N
0224	D4	Code for letter T
0225	00	Not used for this command
0226	01	Page where COUNT routine starts
0227	C0	Location on page where COUNT starts
0228	C5	Code for letter E
0229	D2	Code for letter R
022A	C1	Code for letter A
022B	D3	Code for letter S
022C	C5	Code for letter E
022D	00	Not used for this command
022E	01	Page where ERASE routine starts
022F	E0	Location on page where ERASE starts
0230	00	**End of table marker**

FREE FORMAT CONTROL TABLE

0200	C7	Code for letter G
0201	CF	Code for letter O
0202	00	*End of command word marker*
0203	01	Page where GO routine starts
0204	40	Location on page where GO starts
0205	CC	Code for letter L
0206	C9	Code for letter I
0207	D3	Code for letter S
0208	D4	Code for letter T
0209	00	*End of command word marker*
020A	01	Page where LIST routine starts
020B	60	Location on page where LIST starts
020C	CD	Code for letter M
020D	C5	Code for letter E
020E	C4	Code for letter D
020F	C9	Code for letter I
0210	C1	Code for letter A
0211	CE	Code for letter N
0212	00	*End of command word marker*
0213	01	Page where MEDIAN routine starts

0214	80	Location on page for MEDIAN
0215	C1	Code for letter A
0216	D6	Code for letter V
0217	C7	Code for letter G
0218	00	*End of command word marker*
0219	01	Page where AVG routine starts
021A	A0	Location on page where AVG starts
021B	C3	Code for letter C
021C	CF	Code for letter O
021D	D5	Code for letter U
021E	CE	Code for letter N
021F	D4	Code for letter T
0220	00	*End of command word marker*
0221	01	Page where COUNT STARTS
0222	C0	Location on page where COUNT starts
0223	C5	Code for letter E
0224	D2	Code for letter R
0225	C1	Code for letter A
0226	D3	Code for letter S
0227	C5	Code for letter E
0228	00	*End of command word marker*
0229	01	Page where ERASE starts
022A	E0	Location on page where ERASE starts
022B	00	**End of table marker**

As mentioned before, the lengths of the two tables differ because of the variation in the number of characters for each command name. If, however, all of the names were six characters long, the fixed format table would be shorter than the free format, since the command field name in the free format table would require seven bytes to store each name - six for the name and one for the terminating zero byte.

Another consideration when deciding which format to use is the type of input programming required to enter the commands. There are several different methods that may be used to input and store the command to be searched for in the control table.

One method is to initially clear out the input buffer area by filling it with zero bytes. Then, as each character is entered, it is stored in the input buffer. When a carriage return is entered, the input is terminated and the contents of the input buffer may be used to search for the command. If the command entered does not fill the input buffer, the unused locations will contain zero bytes. This method is best suited for the fixed format, since the input buffer will contain

the same contents as the command name field of the matching command in the control table.

The following routine could be used to clear the input buffer and store the characters in the buffer as discussed above. This routine uses the CLRMEM subroutine in chapter three to clear the input buffer. The INPUT routine that is called must input a character from the input device (such as an ASCII keyboard) and return with the character in the A accumulator. Along with the test for the carriage return, to terminate the input and return, the character count is checked and when the input buffer is full; any additional characters that may be inadvertently entered before the carriage return are ignored. The initial instruction sequence may be used as a control routine to call the individual routines, including the search routine to be presented later.

```
NEXCMD  LDX #INPBFR     Set pointer to start of input buffer
        LDAB #$06       Set clearing counter
        JSR CLRMEM      Clear input buffer area
        BSR INCMND      Fetch command string fm input device
        BSR SRCHFX      Search tbl & perform command input
        BRA NEXCMD      Repeat loop for next command

INCMND  LDX #INPBFR     Set pointer to start of input buffer
        LDAB #$06       Set counter for maximum size of bfr
INCHAR  JSR INPUT       Call routine to input character
        CMPA #$8D       See if character was carriage return
        BNE CHECK       No, continue input routine
        RTS             Yes, return, input complete

CHECK   TSTB            Test character counter for zero
        BEQ INCHAR      If zero, ignore new character
        DECB            Otherwise, decrement value of counter
        STAA X          And store character in buffer
        INX             And advance input buffer pointer
        BRA INCHAR      Loop to fetch next character from I/O
```

Another method of inputting the characters is to leave the input buffer contents as is at the start of the input routine. As each character is received, it is stored in the input buffer. When a carriage return is received, the input is terminated by storing the carriage return in the input buffer and returning. Thus, the input buffer area must be assigned one byte more than the maximum number of characters assigned for a command name. This method is more advantageous for

the free format search routine because it sets up the command entered in a similar format to that used in the command name field of the free format control table entries.

An input routine that operates in this manner is listed next. The only real difference between this routine, labeled INCTRL, and the previous INCMND routine is the instruction sequence that stores the carriage return as the terminating character in the input buffer before returning. Also, one should note the absence of the routine that clears the input buffer before inputting the command. This saves quite a few memory locations. As in the previous listing, the initial instruction sequence is a sample control routine for directing the operation of the command search function.

```
NEXCMD  BSR INCTRL      Fetch command string fm input device
        BSR SRCHFR      Search tbl & perform command input
        BRA NEXCMD      Repeat loop for next command

INCTRL  LDX #INPBFR     Set pointer to start of input buffer
        LDAB #$06       Set cntr for maximum nmbr of chars
INCHAR  JSR INPUT       Call routine to input character
        CMPA #$8D       See if character was a carriage return
        BNE CHECK       If not, check for buffer full
        STAA X          If so, store CR as last character in bfr
        RTS             And return to calling program

CHECK   TSTB            Test character counter for zero
        BEQ INCHAR      If zero, ignore new character
        DECB            Otherwise, decrement value of counter
        STAA X          And store character in buffer
        INX             And advance input buffer pointer
        BRA INCHAR      Loop to fetch next character from I/O
```

The search routine for a fixed format control table, listed next, compares the contents of the input buffer to the command name field on a character-by-character basis. This is done by calling the CPRMEM suboutine, beginning at the CMATCH label, which is presented in chapter three. This subroutine may be included in the SRCHFX routine if it is not used elsewhere in one's program.

If the characters in the input buffer do not match the command name field of an entry in the control table, the NXWORD routine is entered to advance the control table pointer to the start of the next

entry. At this point, the first character of this entry is checked for the zero byte, which indicates the end of the control table. If the zero byte is not found, the routine jumps to the compare routine to check for a match between the new control entry and the input buffer. When the zero byte is encountered, it indicates that the entire table has been searched and no match has been found. The routine then returns to the control routine to initiate a new command entry.

It may be desirable at this point to have the search routine print out a message if no match is found, to inform the operator of the error. This may be done by changing the RTS instruction in the SETNXW routine to a branch or jump instruction, which jumps to a message output routine to print the message before returning to the main control program.

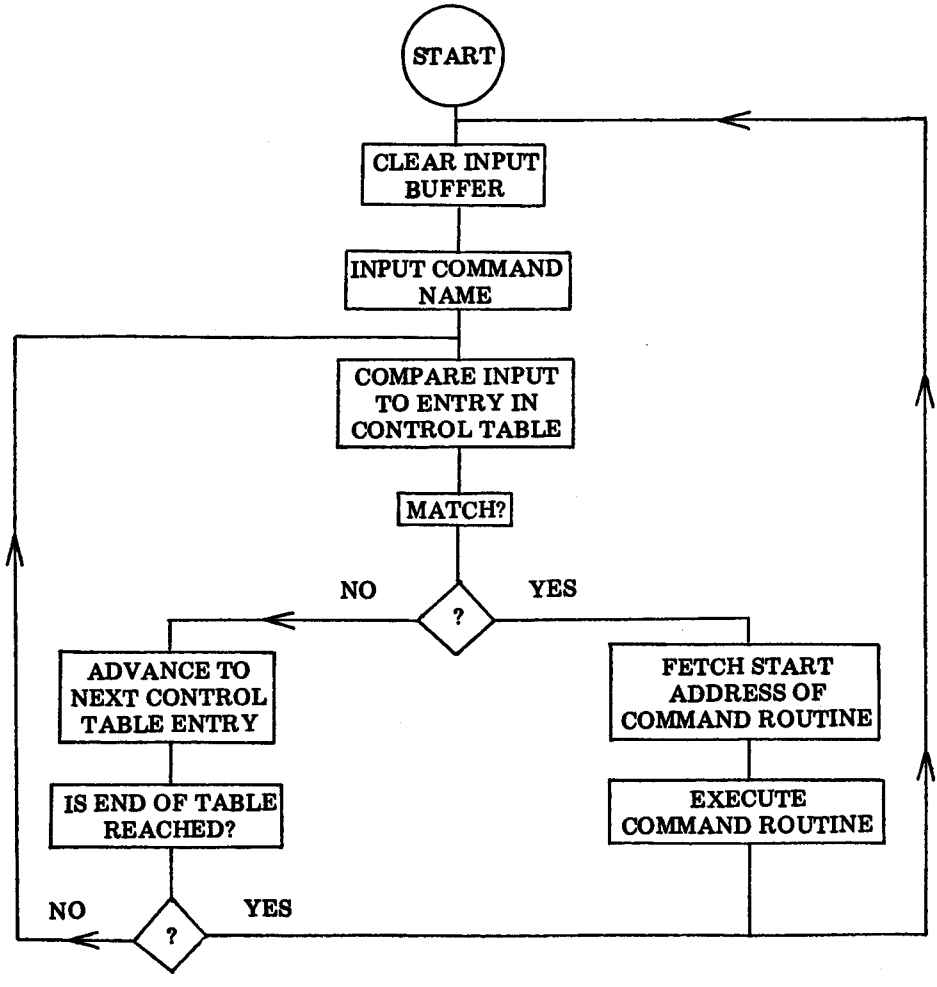
When a match is found, the FOUND routine is entered. This routine takes the address from the address field of the matching control entry and uses it to jump to the command routine by placing the address in the index register and executing a JMP X instruction. After the command routine completes its operations, it may return to the main control program by simply executing a return instruction.

Since this routine compares the entire input buffer against the command name field of the control table entries, it is not necessary for it to test the input buffer or command name for a terminating character. However, a counter must be set to the number of characters in the command name field so that the routine will know when all of the characters have been compared. In this routine, this counter is set to six. If one changes the length of the command name field, this counter must also be changed to reflect the new length.

The listing for this fixed format search routine is presented next, followed by the flow chart. The flow chart also includes the logic flow of the main control routine, previously presented, when used in conjunction with this search routine.

```
SRCHFX   LDX #CMDTBL     Set up pointer to starting addr of table
 INITBF  STX TEMP2       Store in TEMP2
         LDX #INPBFR     Set up pointer to start of input buffer
         STX TEMP1       Store in TEMP1
         LDAB #$06       Set field size counter
CMATCH   JSR CPRMEM      Compare input bfr to table contents
```

```
        BEQ FOUND       Entries equal, process command
NXWORD  DECB            Decrement field size counter
        BEQ SETNXW      End of cntrl field? Branch when found
        INX             Otherwise, advance table pointer
        BRA NXWORD      And loop to look for end of field
SETNXW  INX             At end of control word need to
        INX             Advance pointer over the address field
        INX             To start of next control word field
        TST X           Is 1st character of next field zero byte
        BNE INITBF      If not, continue table search
        RTS             Else, return to command input
FOUND   INX             Advance pointer to command address
        LDX X           Fetch address from command entry
        JMP X           Jump to command routine
```



The free format search routine that follows has the same basic flow of the fixed format routine. That is, it compares the input buffer to an entry in the control table and, if they do not match, it advances the control table pointer to compare the next entry against the input buffer. This continues until either a match is found, or the end of the buffer is reached. But instead of comparing a set number of characters to get a match, this routine compares the contents of the input buffer, up to the terminating carriage return, against each command name field in the control table. If the input buffer and command name field match up to the carriage return in the input buffer, the corresponding location in the command name field is checked for its terminating character, a zero byte. If the zero byte is found, the same FOUND routine as in SRCHFX is entered to fetch the address from the control table entry and jump to the proper command routine. Here again, if the suggested main control routine for the free format table is used, the command routines may return to the main control routine by executing a return instruction. The listing for this search routine is shown next, followed by the flow chart.

```
SRCHFR  LDX #CMDTBL     Set up pointer to starting addr of table
INITBF  STX TEMP2       Store table pointer in TEMP2
        LDX #INPBFR     Set up pointer to start of input buffer
CMATCH  LDAA X          Fetch character from input buffer
        INX             Advance input buffer pointer
        STX TEMP1       Store input buffer pointer
        LDX TEMP2       Fetch pointer to control table
        CMPA #$8D       See if symbol is a carriage return
        BEQ LCHAR       If so, go to last character routine
        CMPA X          See if have match condition in table
        BNE NXWORD      If not, go to next block in table
        INX             Advance input buffer pointer
        STX TEMP2       Save control table pointer
        LDX TEMP1       Fetch pointer to input buffer
        BRA CMATCH      Check next character

LCHAR   TST X           Is end of control field here?
        BEQ FOUND       Yes, found matching control word

NXWORD  TST X           Test for end of control field
        BEQ SETNXW      If so, advance to next block
        INX             Otherwise, advance command pointer
        BRA NXWORD      And continue looking

SETNXW  INX             Advance pointer over the address field
        INX             To start of next control word field
```

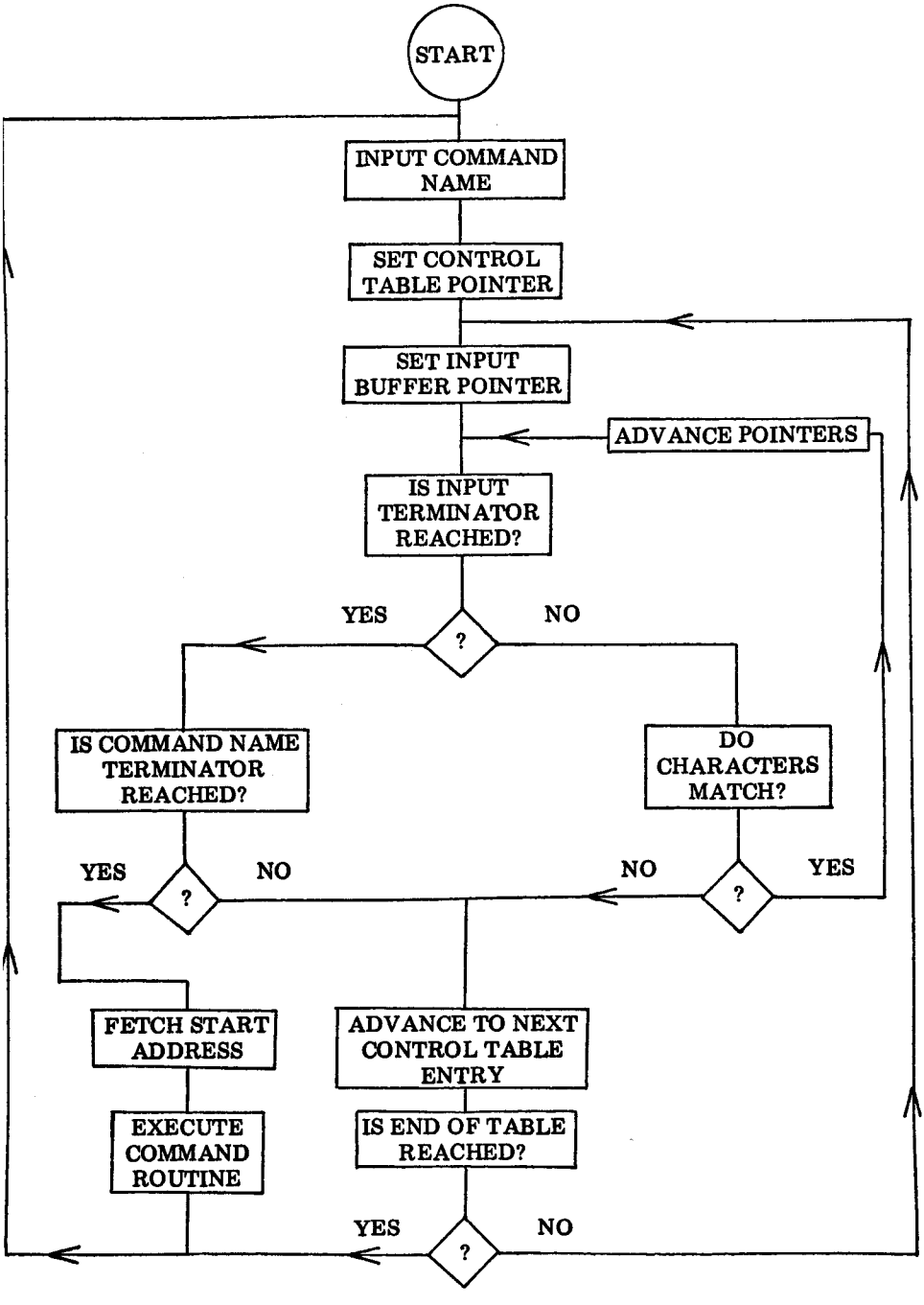
```
        INX
        TST X           Is end of control table reached?
        BNE INITBF      No, loop back to check next command
        RTS             Yes, return if no match is found

FOUND   INX             Advance table pointer to address
        LDX X           Fetch address from command routine
        JMP X           Jump to command routine
```

The process of sorting data into some specified order, or into groups with common attributes, is another of the computer's powerful capabilities. For example, it is often desired to arrange a list of names and addresses in alphabetical order according to the last names. Or, one may want to sort out persons living in a common geographical location by sorting the addresses according to the zip code. In order to program these types of sort functions, the data must first be organized into carefully structured tables.

Proper structuring may be accomplished by creating fields in which specific items are to be located. These fields are set up in a similar manner as the fields in the search table entries. For the sort routine to be discussed, a fixed format table will be used. This table will contain a list of names which are to be arranged in alphabetical order. For illustrative purposes, the following names will be used as table entries.

BROWN, L.R.
DAMATO, P. J.
ANDERSON, B.
DARBY, P.
MATTOX, R.T.
MATTHEWS, K.D.
JONES, A.M.

Each element of the names in this list must have a specific field assigned to it. Looking at the last names, one may observe that the longest name is eight characters long. However, there are many names that use more than eight characters. Therefore, a field of fourteen bytes for the last name will be used to accommodate the longer names. One byte fields will be set up for each of the initials. This makes a total of sixteen bytes for each entry. The delimiters (the comma and period) are not assigned any location in the tables since a fixed format is to be used and the inclusion of these punctuation marks would only serve to take up table space. The delimiters are used though, when entering the names to be stored in the table.


START
INPUT COMMAND NAME
SET CONTROL TABLE POINTER
SET INPUT BUFFER POINTER
ADVANCE POINTERS
IS INPUT TERMINATOR REACHED?
YES
?
NO
IS COMMAND NAME TERMINATOR REACHED?
DO CHARACTERS MATCH?
YES
?
NO
NO
?
YES
FETCH START ADDRESS
ADVANCE TO NEXT CONTROL TABLE ENTRY
EXECUTE COMMAND ROUTINE
IS END OF TABLE REACHED?
YES
?
NO

The following program is one possible means of entering the names into the table in the properly formatted fields. This routine accepts the names as keyboard entries in the format illustrated above. Each field is accepted and stored in its proper location in the table for that entry. A name input is terminated by entering a carriage return. The delimiters (comma and period) terminate a field entry and advance the pointer to the next field in the entry. If a field is not filled up by the name, such as a last name with less than fourteen characters or one with no middle initial, the unused locations in the field are filled with zero bytes. After the final name has been entered, the operator may input an asterisk to indicate the completion of the input operation.

The character input is expected to be ASCII characters. As pointed out in the chapter on conversion codes, the ASCII character set is a well-ordered set of character codes. The letters A through Z are coded in consecutive order from 301 to 332 (octal), and the numbers 0 through 9 are coded in order from 260 to 271 (octal). This is especially useful when sorting into alphabetical or numerical order, since the lower the code for the character, the lower it will place in the entry in the sorted list.

Before listing this routine, several comments about its operation must be given. First, the characters are received by calling a routine referred to by the label INPUT. This routine must be provided by the user to accept characters from the input device associated with one's system. This routine must return the ASCII code for the character inputted in the accumulator upon returning. As a visual indication for the operator to verify the characters inputted, this INPUT routine should also output the characters received to the system display device before returning. The contents of the index register must not be altered by this routine. If the index register must be used, to point to a conversion table or whatever, it should be saved and then restored before returning. This INPUT routine should also check for the receipt of a carriage return character. When received, the input routine should also output a line feed character to the printer device, since this is not provided for in the name input routine. (A sample input routine that utilizes the MIKBUG** is listed immediately following the name input routine.)

Before this routine is called, the table area must be properly set up. This is accomplished by storing a zero byte in the first location in the table. The zero byte is used to indicate the location for storing

the next name inputted. Initially, this byte must be set up at the start of the table by the calling program. Then, as each name is entered into the table, the zero byte is moved up to the location immediately following the last name entered in the table. Before each name input is initiated, the location of this byte in the table is checked. If the zero byte is not found within the limits of the table area, it is assumed that the table is filled. At this point, a routine, to be supplied by the user and referred to here as TOMUCH, would be entered. TOMUCH should output a message to the operator indicating that the table is filled. The limits of the table area in this sample routine begin at page 04 location 000, and end at page 07 location 377. Thus, when the table pointer is advanced to the start of page 10, the table is full.

After the zero byte is found and the program is ready to accept a name input, it calls the input routine to fetch the first character. There are a number of special codes checked when this first character is entered. One is an asterisk, which is to be inputted when the list contains all of the names desired. When this character is received, the routine returns to the calling program, which may then call upon the sort routine to sort the names into alphabetical order. The other special codes checked for are the carriage return, comma, or period. If any of these codes are received as the first character, they are ignored. This is because they indicate either the end of an entry or field, which would not be valid at this time since there are no characters stored as yet for this name.

Once a valid character is entered for the first character, the characters that follow are checked for a comma or carriage return. If the comma is received, the remainder of the last name field is filled with zero bytes, and the portion of the routine that accepts the initials is entered. If a carriage return is received, the remainder of the entire entry is filled with zeros, and a new name input is initiated. If the character entered is neither of these two characters, it is entered as the next character in the last name, up to the fourteenth character. If more than fourteen characters are entered for the last name, the excess characters are simply ignored.

If, when the first initial entry is to be entered, a carriage return is received, the two initial fields are zeroed and a new name input is begun. If a comma is received, it is ignored. Otherwise, the character is stored as the first initial and the routine jumps to input the second initial. When the second initial is to be entered, receipt of a period is

ignored, and a carriage return results in a zero byte being stored for the second initial. Any other input is stored as the second initial. The routine then checks for a full table, and, if not filled, begins a new name input sequence.

```
ACCEPT  LDX #SRTTBL     Initialize sort table pointer
NOTFND  TST X           Is pointer to end of storage area?
        BEQ FNDEND      Yes, begin input of name
        LDAB #$10       To start of next entry by incrementing
ADVNCE  INX             It $10 times
        DECB            Decrement counter
        BNE ADVNCE      Counter ≠ 0, continue advancing
        BSR CKPAGS      See if out of boundary (04 - 07)
        BRA NOTFND      No, keep looking for end of strg area

FNDEND  LDAB #$0E       Set up last name field counter
        JSR INPUT       And fetch character from input
        CMPA #$AA       Check for * code (finished indicator)
        BNE NOTDON      Proceed if not * code
        CLR X           Store ending marker at start of next
        RTS             Entry and exit subroutine
NOTDON  CMPA #$8D       Test for carriage return
        BEQ FNDEND      Ignore if first character in field
        CMPA #$AE       Test for (.) code
        BEQ FNDEND      Ignore if first character in field
        CMPA #$AC       Test for (,) code
        BEQ FNDEND      Ignore if first character in field
STRCHR  STAA X          If none of above, store char in field
        INX             Advance storage pointer
        DECB            Decrement field counter
        BEQ FULFLD      If 0, field is full
        JSR INPUT       Otherwise, fetch next input
        CMPA #$8D       Test for carriage return
        BEQ HAVECR      Finish entry if have carriage return
        CMPA #$AC       Test for comma
        BEQ HAVECM      Finished last name field if comma
        BRA STRCHR      Jump to store character in field
HAVECR  INCB            Increment counter twice to allow
        INCB            Clearing of initial fields too
        CLR X           C/R received, clear rest of entry
        INX             Advance storage pointer
        DECB            Decrement entry counter
        BNE HAVECR+$2   If not 0, continue clearing
        BSR CKPAGS      See if out of boundary
        CLR X           Not out, clear first character
        BRA NOTFND      Begin process for new entry
HAVECM  CLR X           If comma, clear rest of last name fld
        INX             Advance storage pointer
        DECB            Decrement field counter
```

```
          BNE HAVECM      If not 0, continue clearing
FULFLD    JSR INPUT       Get character for first initial
          CMPA #$AC       Test for comma
          BEQ FULFLD      Ignore comma at this point
          CMPA #$8D       Test for carriage return
          BNE SAVIN1      If not carriage return, store character
          CLRA            If carriage return, store 00 byte in
          STAA X          Both initial fields
          INX             Advance storage pointer, then
          BRA SAVIN2      Follow branch to clear 2nd initial

SAVIN1    STAA X          Store first initial in first initial field
          INX             Advance storage pointer
INITF2    JSR INPUT       Look for second initial
          CMPA #$AE       Check for period
          BEQ INITF2      Ignore a period
          CMPA #$8D       Check for a carriage return
          BNE SAVIN2      If not C/R, store as second initial
          CLRA            If C/R, place 00 byte in memory
SAVIN2    STAA X          Store character or 00 byte substitute
          INX             Advance pointer to next entry
          BSR CKPAGS      Go check if out of boundary
          BRA FNDEND      If not, process next input

CKPAGS    CPX #$0800      Test pointer for exceeding boundary
          BEQ TOMUCH      Display message if table filled
          RTS             Otherwise, return to continue input

INPUT     JSR $E1AC       Call MIKBUG** keyboard input
          ORAA #$80       Set MSB of ASCII character
          CMPA #$8D       Is character a carriage return?
          BNE INRET     * No, return to calling program
          LDAA #$8A       Set up line feed code
          JSR $E1D1       And output to printer
          LDAA #$8D       Restore carriage return code
INRET     RTS             Return to calling program
```

It may be desired to provide some kind of entry correction capability to this main input routine. One way to accomplish this might be to designate another special code that, when entered, would cause the program to reset the table pointer to the start of the current entry and initiate a new name input. The routine listed below may be used by the input routine to check for a control O character (8F hexadecimal). This routine checks for the control O and, if entered, resets the table pointer, essentially erasing the current name input from the table, and jumps to the start of the name input sequence as the FNDEND label. Otherwise, it simply returns to

continue the input. The instructions in the INPUT routine marked with an asterisk would be changed to branch to this routine rather than the return instruction.

```
CHKRUB   CMPA  #$8F        Check for control O
         BEQ RESET         If control O, start entry again
         RTS               Otherwise, return to check character
RESET    INS               Advance stack pointer to
         INS               Eliminate current return address
         STX TEMP1         Store pointer in memory and
         LDAA  #$F0        Reset to start of entry by masking
         ORAA TEMP1+$1     Off four LSB's of low portion
         STAA TEMP1+$1     Restore low portion of pointer
         LDX TEMP1         And restore back to index register
         JMP FNDEND        Begin name entry again
```

Now that one has defined the format for storing the data, and developed a means of entering it, a routine may be written to sort the data as desired. The main objective of the sort routine is to examine the contents of the field (or fields) that contains the information pertinent to the sort operation, and rearrange the table contents into the desired order or groups. There are a number of techniques used to do this, the choice of which generally depends on the type of data and sorting operation to be performed. The sort routine presented next arranges the table contents into alphabetical order by using what is referred to as a "ripple" sorting technique.

The term ripple is derived from the manner in which the routine moves through the table to sort the entries into alphabetical order. Beginning with the first entry (N), the sort routine compares it to the next entry (N+1) in the table. If the first entry is lower in alphabetical order than the second, the two entries are left as is, and the routine advances to check the order of the second entry (new N) against the third (new N+1). If the first entry is greater than the second, the routine will swap the two entries so that the entry that was initially the second entry would now be the first.

As the procedure continues, if the Nth entry is found to be greater than the N+1 entry, the two entries are exchanged in the table. Then, rather than advancing to the next entry, the routine backs up to compare the N-1 entry against the new N entry. This is because if the initial N+1 entry was lower than the N entry, it may also be lower

than the N-1 entry. Therefore, the routine will continue to transfer the lower entry down through the table until the entry before it is lower in alphabetical order, or the entry is moved to the beginning of the table. The routine then starts back up through the table once again. This type of movement up and down through the table gives a "ripple" effect as the routine compares and shifts the entries around.

The operation of the sort routine may be illustrated by examining its procedure for arranging the sample list of names, given at the start of this section, into alphabetical order. The routine initially compares the first entries and finds the order to be correct, since the B in BROWN comes before the D in DAMATO. When the next pair of entries is compared, however, it is found that ANDERSON should be before DAMATO. The routine will therefore shift the second and third entries around, as illustrated in the table below.

BROWN, L. R.
ANDERSON, B.
DAMATO, P. J.
DARBY, P.
MATTOX, R. T.
MATTHEWS, K. D.
JONES, A. M.

Now that the second and third entries are in the proper order with respect to each other, the routine backs up to compare the first entry against the second. The second entry is now found to be less than the first (ANDERSON should come before BROWN), so the routine swaps these two entries, as shown next, and begins comparing the entries once again, starting with the first two entries.

ANDERSON, B.
BROWN, L. R.
DAMATO, P. J.
DARBY, P.
MATTOX, R. T.
MATTHEWS, K. D.
JONES, A. M.

On this pass through the table, the routine will proceed all the way up to MATTOX, R. T. before finding another entry out of order.

Note that in comparing MATTOX, R. T. and MATTHEWS, K. D., the routine must work up to the fifth character in the last names to determine the proper order of the two names. If the last names were the same, it must go up to the initials to check whether the two entries are in order. Upon finding these two names in the wrong order, the routine will exchange them.

ANDERSON, B.
BROWN, L. R.
DAMATO, P. J.
DARBY, P.
MATTHEWS, K. D.
MATTOX, R. T.
JONES, A. M.

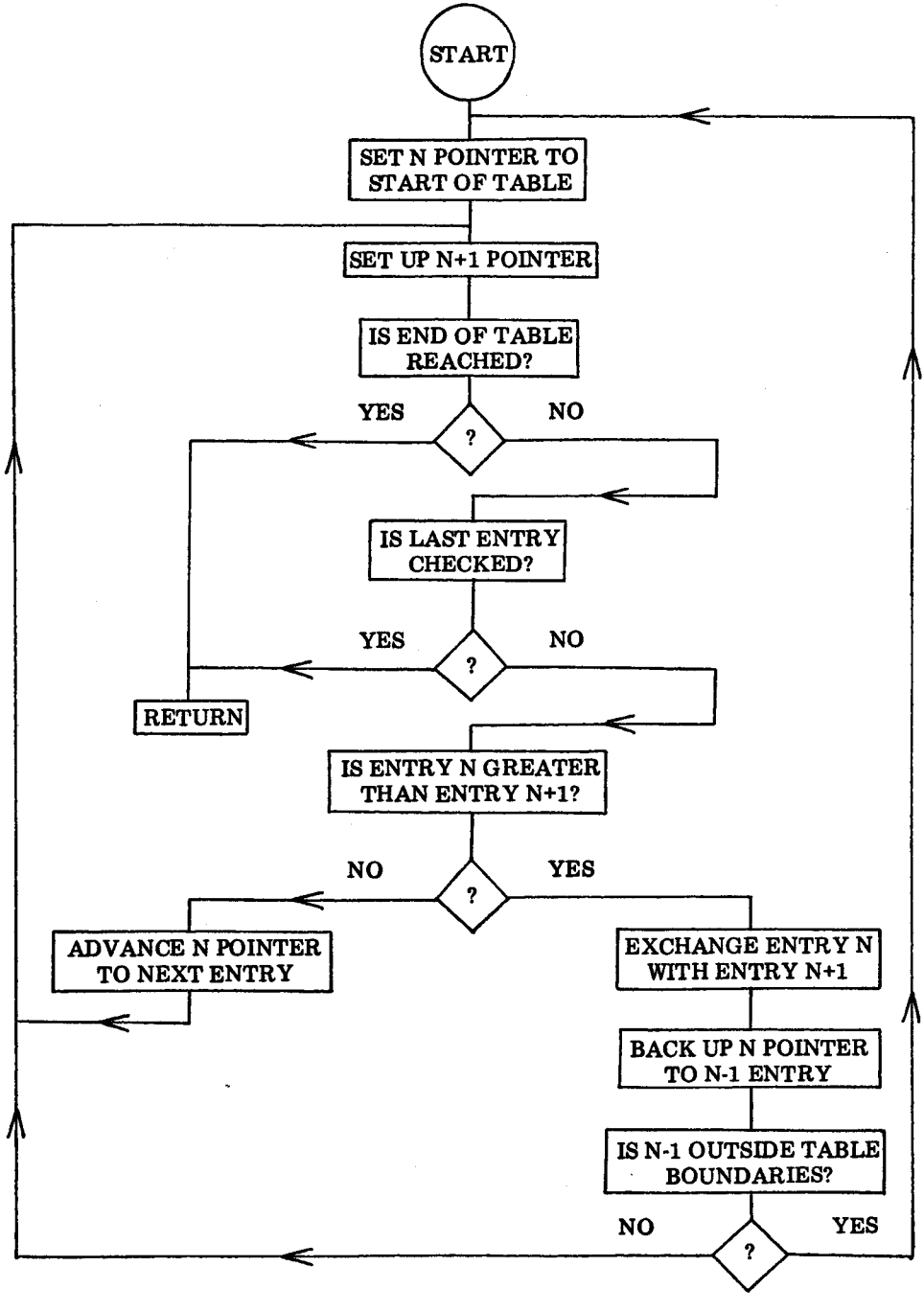
The routine then backs up in order to compare DARBY, P. and MATTHEWS, K. D. Finding these to be in order, it moves forward again until MATTOX, R. T. is compared to JONES, A. M. These two entries are swapped and the routine again backs up and compares MATTHEWS, K. D. to JONES, A. M. which it finds must also be swapped. Finally, after determining that DARBY, P. and JONES, A. M. are in the right order, the routine advances until the end of the table is reached. The resulting table will contain the names in the following order.

ANDERSON, B.
BROWN, L. R.
DAMATO, P. J.
DARBY, P.
JONES, A. M.
MATTHEWS, K. D.
MATTOX, R. T.

A routine that performs this type of ripple sorting is listed next, along with a flow chart of its operations. In checking for the start and end of the allowable table, this routine assumes that the table begins at page 04 location 00, and ends at page 07 location FF. If the entire table is not filled, the last entry must be followed by a zero byte, as discussed previously. The instruction sequence used here to compare the two entries is similar to the CPRMEM routine presented in chapter three. One should make special note of the

fact that both the compare and exchange portions of this routine use the index register plus a displacement to point to the N+1 entry. This is possible because of the fixed length assigned to the entry. As one can see, this eliminates the necessity to swap pointers between the index register and temporary storage locations.

```
SORT    LDX #SRTTBL     Set pointer to start of table
INITBK  CPX #$07F0      Check for end of table
        BEQ SRTRET      End reached? Sort operation complete
        LDAB #$10       Set entry length counter
        TST $10,X       Is first character of next entry =0?
        BNE CKNEXT      No, compare next pair of entries
SRTRET  RTS             Yes, sort operation complete, return

CKNEXT  STX TEMP1       Store pointer to N entry
        LDAA X          Fetch character from N entry
        CMPA $10,X      Compare N to N+1
        BNE CKGTLT      Not equal, ck for greater or less than
        INX             Advance N pointer
        DECB            Characters equal, decrement entry cntr
        BNE CKNEXT+$2   Counter ≠ 0, continue checking
        BRA INITBK      Equals 0, entries match, try next entry
CKGTLT  BLS FINEND      N < N+1, order O.K., try next entry
        LDAB #$10       N > N+1, exchange entries, set cntr
        LDX TEMP1       Fetch N pointer
NOTYET  LDAA X          Fetch character from N entry
        STAA TEMP2      Save N character temporarily
        LDAA #$10,X     Fetch character from N+1
        STAA X          Store character from N+1 in N
        LDAA TEMP2      Fetch N char from temp storage
        STAA #$10,X     Store character from N in N+1
        INX             Advance N pointer
        DECB            Decrement entry length counter
        BNE NOTYET      ≠0, continue exchanging
        LDAB #$20       Set up counter for backing up pointer
NMINUS  DEX             Back up N pointer two full entries
        DECB            Decrement counter
        BNE NMINUS      ≠0, continue backing up pointer
        CPX  $0400      Is pointer below start of table?
        BLT SORT        Yes, reset pointer and begin again
        BRA INITBK      No, check starting at N+1 entry

FINEND  INX             Advance N pointer
        DECB            Decrement entry length counter
        BNE FINEND      ≠0, continue advancing pointer
        BRA INITBK      Pointer at next entry, continue sort
```


START
SET N POINTER TO START OF TABLE
SET UP N+1 POINTER
IS END OF TABLE REACHED?
YES
?
NO
IS LAST ENTRY CHECKED?
YES
?
NO
RETURN
IS ENTRY N GREATER THAN ENTRY N+1?
NO
?
YES
ADVANCE N POINTER TO NEXT ENTRY
EXCHANGE ENTRY N WITH ENTRY N+1
BACK UP N POINTER TO N-1 ENTRY
IS N-1 OUTSIDE TABLE BOUNDARIES?
NO
?
YES

This method of sorting may be aided in a number of ways to increase the efficiency of its operation. For example, the name input routine could be revised to separate the table into several sections, one for names beginning with the letters A through J, another for K through R, and another for S through Z. As each name is entered, the first letter could be checked, and the name stored in the proper section of the table.

Another possibility is to revise the ripple sequence in the following manner. When a name is found to be out of alphabetical order, the start address of the current N+1 entry could be saved. Then, after the entry is backed up to the proper location in the table, the sort may resume by recalling the saved N+1 address and using it as the N address of the next entry to compare. This would avoid the time consuming process of retracing the sort up through the section already known to be in the proper order.

The sort function could also be revised to limit itself to the contents of just one field in an entry. By setting the pointers and field length counter to a specific field within each entry, the sort operation could arrange the entries according to some classification, such as the zip code of an address, or a special code set up by the programmer to classify each entry.

The techniques and routines discussed in this chapter may be utilized to create rather sophisticated programs designed specifically to fill one's requirements. By combining these with other programming functions presented in this book, one may develop programs that give the computer the capability to perform various operations for a wide variety of applications.

APPENDIX A

6800 INSTRUCTION SET

This table presents the entire instruction set of the 6800 CPU. The first column contains the mnemonic used for each instruction. The machine code, presented as hexadecimal digits, is given in the second column. For the instructions that use one or more of the possible addressing modes, the third column indicates the mode for the machine code of that row. The fourth and fifth columns indicate the number of bytes required for the instruction cycles and the number of machine cycles, respectively. The final column indicates the page in chapter one in which the instruction is defined.

MNEMONIC	MACHINE CODE	ADDRESSING MODE	NO. OF BYTES	NO. OF CYCLES	PAGE REF.
ABA	1B		1	2	1-23
ADCA	89	IMMEDIATE	2	2	1-22
ADCA	99	DIRECT	2	3	1-22
ADCA	A9	INDEXED	2	5	1-22
ADCA	B9	EXTENDED	3	4	1-22
ADCB	C9	IMMEDIATE	2	2	1-22
ADCB	D9	DIRECT	2	3	1-22
ADCB	E9	INDEXED	2	5	1-22
ADCB	F9	EXTENDED	3	4	1-22
ADDA	8B	IMMEDIATE	2	2	1-22
ADDA	9B	DIRECT	2	3	1-22
ADDA	AB	INDEXED	2	5	1-22
ADDA	BB	EXTENDED	3	4	1-22
ADDB	CB	IMMEDIATE	2	2	1-22
ADDB	DB	DIRECT	2	3	1-22
ADDB	EB	INDEXED	2	5	1-22
ADDB	FB	EXTENDED	3	4	1-22
ANDA	84	IMMEDIATE	2	2	1-28
ANDA	94	DIRECT	2	3	1-28
ANDA	A4	INDEXED	2	5	1-28
ANDA	B4	EXTENDED	3	4	1-28
ANDB	C4	IMMEDIATE	2	2	1-28
ANDB	D4	DIRECT	2	3	1-28
ANDB	E4	INDEXED	2	5	1-28
ANDB	F4	EXTENDED	3	4	1-28
ASL	68	INDEXED	2	7	1-34
ASL	78	EXTENDED	3	6	1-34
ASLA	48		1	2	1-34
ASLB	58		1	2	1-34
ASR	67	INDEXED	2	7	1-35

ASR	77	EXTENDED	3	6	1-35
ASRA	47		1	2	1-35
ASRB	57		1	2	1-35
BCC	24	RELATIVE	2	4	1-39
BCS	25	RELATIVE	2	4	1-39
BEQ	27	RELATIVE	2	4	1-39
BGE	2C	RELATIVE	2	4	1-40
BGT	2E	RELATIVE	2	4	1-40
BHI	22	RELATIVE	2	4	1-39
BITA	85	IMMEDIATE	2	2	1-30
BITA	95	DIRECT	2	3	1-30
BITA	A5	INDEXED	2	5	1-30
BITA	B5	EXTENDED	3	4	1-30
BITB	C5	IMMEDIATE	2	2	1-30
BITB	D5	DIRECT	2	3	1-30
BITB	E5	INDEXED	2	5	1-30
BITB	F5	EXTENDED	3	4	1-30
BLE	2F	RELATIVE	2	4	1-40
BLS	23	RELATIVE	2	4	1-39
BLT	2D	RELATIVE	2	4	1-40
BMI	2B	RELATIVE	2	4	1-39
BNE	26	RELATIVE	2	4	1-39
BPL	2A	RELATIVE	2	4	1-39
BRA	20	RELATIVE	2	4	1-38
BSR	8D	RELATIVE	2	8	1-42
BVC	28	RELATIVE	2	4	1-39
BVS	29	RELATIVE	2	4	1-39
CBA	11		1	2	1-26
CLC	0C		1	2	1-31
CLI	0E		1	2	1-32
CLR	6F	INDEXED	2	7	1-20
CLR	7F	EXTENDED	3	6	1-20
CLRA	4F		1	2	1-15
CLRB	5F		1	2	1-15
CLV	0A		1	2	1-32
CMPA	81	IMMEDIATE	2	2	1-26
CMPA	91	DIRECT	2	3	1-26
CMPA	A1	INDEXED	2	5	1-26
CMPA	B1	EXTENDED	3	4	1-26
CMPB	C1	IMMEDIATE	2	2	1-26
CMPB	D1	DIRECT	2	3	1-26
CMPB	E1	INDEXED	2	5	1-26
CMPB	F1	EXTENDED	3	4	1-26
COM	63	INDEXED	2	7	1-21
COM	73	EXTENDED	3	6	1-21
COMA	43		1	2	1-16
COMB	53		1	2	1-16
CPX	8C	IMMEDIATE	3	3	1-26
CPX	9C	DIRECT	2	4	1-26

CPX	AC	INDEXED	2	6	1-26
CPX	BC	EXTENDED	3	5	1-26
DAA	19		1	2	1-23
DEC	6A	INDEXED	2	7	1-21
DEC	7A	EXTENDED	3	6	1-21
DECA	4A		1	2	1-15
DECB	5A		1	2	1-15
DES	34		1	4	1-19
DEX	09		1	4	1-18
EORA	88	IMMEDIATE	2	2	1-30
EORA	98	DIRECT	2	3	1-30
EORA	A8	INDEXED	2	5	1-30
EORA	B8	EXTENDED	3	4	1-30
EORB	C8	IMMEDIATE	2	2	1-30
EORB	D8	DIRECT	2	3	1-30
EORB	E8	INDEXED	2	5	1-30
EORB	F8	EXTENDED	3	4	1-30
INC	6C	INDEXED	2	7	1-20
INC	7C	EXTENDED	3	6	1-20
INCA	4C		1	2	1-15
INCB	5C		1	2	1-15
INS	31		1	4	1-19
INX	08		1	4	1-17
JMP	6E	INDEXED	2	4	1-41
JMP	7E	EXTENDED	3	3	1-41
JSR	AD	INDEXED	2	8	1-42
JSR	BD	EXTENDED	3	9	1-42
LDAA	86	IMMEDIATE	2	2	1-13
LDAA	96	DIRECT	2	3	1-13
LDAA	A6	INDEXED	2	5	1-13
LDAA	B6	EXTENDED	3	4	1-13
LDAB	C6	IMMEDIATE	2	2	1-13
LDAB	D6	DIRECT	2	3	1-13
LDAB	E6	INDEXED	2	5	1-13
LDAB	F6	EXTENDED	3	4	1-13
LDS	8E	IMMEDIATE	3	3	1-18
LDS	9E	DIRECT	2	4	1-18
LDS	AE	INDEXED	2	6	1-18
LDS	BE	EXTENDED	3	5	1-18
LDX	CE	IMMEDIATE	3	3	1-17
LDX	DE	DIRECT	2	4	1-17
LDX	EE	INDEXED	2	6	1-17
LDX	FE	EXTENDED	3	5	1-17
LSR	64	INDEXED	2	7	1-35
LSR	74	EXTENDED	3	6	1-35
LSRA	44		1	2	1-35

LSRB	54		1	2	1-35
NEG	60	INDEXED	2	7	1-21
NEG	70	EXTENDED	3	6	1-21
NEGA	40		1	2	1-16
NEGB	50		1	2	1-16
NOP	01		1	2	1-37
ORAA	8A	IMMEDIATE	2	2	1-29
ORAA	9A	DIRECT	2	3	1-29
ORAA	AA	INDEXED	2	5	1-29
ORAA	BA	EXTENDED	3	4	1-29
ORAB	CA	IMMEDIATE	2	2	1-29
ORAB	DA	DIRECT	2	3	1-29
ORAB	EA	INDEXED	2	5	1-29
ORAB	FA	EXTENDED	3	4	1-29
PSHA	36		1	4	1-14
PSHB	37		1	4	1-14
PULA	32		1	4	1-14
PULB	33		1	4	1-14
ROL	69	INDEXED	2	7	1-36
ROL	79	EXTENDED	3	6	1-36
ROLA	49		1	2	1-36
ROLB	59		1	2	1-36
ROR	66	INDEXED	2	7	1-36
ROR	76	EXTENDED	3	6	1-36
RORA	46		1	2	1-36
RORB	56		1	2	1-36
RTI	3B		1	10	1-44
RTS	39		1	5	1-43
SBA	10		1	2	1-25
SBCA	82	IMMEDIATE	2	2	1-24
SBCA	92	DIRECT	2	3	1-24
SBCA	A2	INDEXED	2	5	1-24
SBCA	B2	EXTENDED	3	4	1-24
SBCB	C2	IMMEDIATE	2	2	1-24
SBCB	D2	DIRECT	2	3	1-24
SBCB	E2	INDEXED	2	5	1-24
SBCB	F2	EXTENDED	3	4	1-24
SEC	0D		1	2	1-31
SEI	0F		1	2	1-32
SEV	0B		1	2	1-32
STAA	97	DIRECT	2	4	1-14
STAA	A7	INDEXED	2	6	1-14
STAA	B7	EXTENDED	3	5	1-14
STAB	D7	DIRECT	2	4	1-14
STAB	E7	INDEXED	2	6	1-14
STAB	F7	EXTENDED	3	5	1-14

STS	9F	DIRECT	2	5	1-18
STS	AF	INDEXED	2	7	1-18
STS	BF	EXTENDED	3	6	1-18
STX	DF	DIRECT	2	5	1-17
STX	EF	INDEXED	2	7	1-17
STX	FF	EXTENDED	3	6	1-17
SUBA	80	IMMEDIATE	2	2	1-24
SUBA	90	DIRECT	2	3	1-24
SUBA	A0	INDEXED	2	5	1-24
SUBA	B0	EXTENDED	3	4	1-24
SUBB	C0	IMMEDIATE	2	2	1-24
SUBB	D0	DIRECT	2	3	1-24
SUBB	E0	INDEXED	2	5	1-24
SUBB	F0	EXTENDED	3	4	1-24
SWI	3F		1	12	1-44
TAB	16		1	2	1-13
TAP	06		1	2	1-33
TBA	17		1	2	1-13
TPA	07		1	2	1-33
TST	6D	INDEXED	2	7	1-27
TST	7D	EXTENDED	3	6	1-27
TSTA	4D		1	2	1-27
TSTB	5D		1	2	1-27
TSX	30		1	4	1-19
TXS	35		1	4	1-19
WAI	3E		1	9	1-43

APPENDIX B

OCTAL TO HEXIDECIMAL

	0	1	2	3	4	5	6	7
00	0	1	2	3	4	5	6	7
01	8	9	A	B	C	D	E	F
02	10	11	12	13	14	15	16	17
03	18	19	1A	1B	1C	1D	1E	1F
04	20	21	22	23	24	25	26	27
05	28	29	2A	2B	2C	2D	2E	2F
06	30	31	32	33	34	35	36	37
07	38	39	3A	3B	3C	3D	3E	3F
10	40	41	42	43	44	45	46	47
11	48	49	4A	4B	4C	4D	4E	4F
12	50	51	52	53	54	55	56	57
13	58	59	5A	5B	5C	5D	5E	5F
14	60	61	62	63	64	65	66	67
15	68	69	6A	6B	6C	6D	6E	6F
16	70	71	72	73	74	75	76	77
17	78	79	7A	7B	7C	7D	7E	7F
20	80	81	82	83	84	85	86	87
21	88	89	8A	8B	8C	8D	8E	8F
22	90	91	92	93	94	95	96	97
23	98	99	9A	9B	9C	9D	9E	9F
24	A0	A1	A2	A3	A4	A5	A6	A7
25	A8	A9	AA	AB	AC	AD	AE	AF
26	B0	B1	B2	B3	B4	B5	B6	B7
27	B8	B9	BA	BB	BC	BD	BE	BF
30	C0	C1	C2	C3	C4	C5	C6	C7
31	C8	C9	CA	CB	CC	CD	CE	CF
32	D0	D1	D2	D3	D4	D5	D6	D7
33	D8	D9	DA	DB	DC	DD	DE	DF
34	E0	E1	E2	E3	E4	E5	E6	E7
35	E8	E9	EA	EB	EC	ED	EE	EF
36	F0	F1	F2	F3	F4	F5	F6	F7
37	F8	F9	FA	FB	FC	FD	FE	FF

APPENDIX C

HEXIDECIMAL TO DECIMAL

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
10	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
20	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
30	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
40	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
50	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
60	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

APPENDIX D

ASCII CHARACTER SET

CHARACTERS SYMBOLIZED	HEXA REP	CHARACTERS SYMBOLIZED	HEXA REP
A	C1	!	A1
B	C2	"	A2
C	C3	#	A3
D	C4	$	A4
E	C5	%	A5
F	C6	&	A6
G	C7	'	A7
H	C8	(	A8
I	C9	)	A9
J	CA	*	AA
K	CB	+	AB
L	CC	,	AC
M	CD	-	AD
N	CE	.	AE
O	CF	/	AF
P	D0	0	B0
Q	D1	1	B1
R	D2	2	B2
S	D3	3	B3
T	D4	4	B4
U	D5	5	B5
V	D6	6	B6
W	D7	7	B7
X	D8	8	B8
Y	D9	9	B9
Z	DA	:	BA
[	DB	;	BB
\	DC	<	BC
]	DD	=	BD
↑	DE	>	BE
←	DF	?	BF
SPACE	A0	@	C0
CAR RET	8D	RUBOUT	FF
LINE FEED	8A	CONTROL O	8F

APPENDIX E

BAUDOT CHARACTER SET

CHARACTERS LC	CHARACTERS UC	5 LEVEL CODE BIT POSITION 5 4 3 2 1	HEXA CODES LC	HEXA CODES UC
A	-	0 0 0 1 1	03	23
B	?	1 1 0 0 1	19	39
C	:	0 1 1 1 0	0E	2E
D	$	0 1 0 0 1	09	29
E	3	0 0 0 0 1	01	21
F	!	0 1 1 0 1	0D	2D
G	&	1 1 0 1 0	1A	3A
H	#	1 0 1 0 0	14	34
I	8	0 0 1 1 0	06	26
J	'	0 1 0 1 1	0B	2B
K	(	0 1 1 1 1	0F	2F
L	)	1 0 0 1 0	12	32
M	.	1 1 1 0 0	1C	3C
N	,	0 1 1 0 0	0E	2C
O	9	1 1 0 0 0	18	38
P	0	1 0 1 1 0	16	36
Q	1	1 0 1 1 1	17	37
R	4	0 1 0 1 0	0A	2A
S	BELL	0 0 1 0 1	05	25
T	5	1 0 0 0 0	10	30
U	7	0 0 1 1 1	07	27
V	;	1 1 1 1 0	1E	3E
W	2	1 0 0 1 1	13	33
X	/	1 1 1 0 1	1D	3D
Y	6	1 0 1 0 1	15	35
Z	"	1 0 0 0 1	11	31
SPACE		0 0 1 0 0	04	04
CAR. RET.		0 1 0 0 0	08	08
LINE FEED		0 0 0 1 0	02	02
NULL		0 0 0 0 0	00	00
FIGURES		1 1 0 1 1	1B	1B
LETTERS		1 1 1 1 1	1F	1F

APPENDIX F

RELOCATABLE FLOATING POINT PROGRAM

The floating point program presented in chapter five has been assembled as a relocatable program and is presented below as a memory dump. The left hand column contains the location of the first memory byte on that line. Each row of data indicates the contents of sixteen memory locations. The symbol table that immediately follows this memory dump indicates the location of the instruction referenced by that symbol within this dump.

The addresses given below are strictly for reference purposes. Since the program is relocatable, it may be loaded into any continuous block of memory exactly as presented here. Also, the program is split into two halves. The first half contains the input and output routines; the second half contains the floating point arithmetic routines. Thus, if one desires to use only the floating point arithmetic routines, the first half of the dump may be discarded, and the second half (beginning at 0403) may be used.

In order to make this program relocatable, the jump and jump to subroutine instructions called out in the listing in chapter five have been changed to branch and branch to subroutine instructions. Also, numerous "islands" have been added to inter-connect the various routines. These islands are indicated by the underlined locations. The ECHO and INPUT routines (which utilize the MIKBUG** I/O routines) are included in this dump. If one's system does not use these routines for I/O, the appropriate I/O driver routines should be added. Finally, the stack pointer is not set up by this program. Therefore, before jumping to the FPCONT routine, the stack pointer should be initialized to a location that will allow for the storage of at least five return addresses.

0100	7F	00	1B	7D	00	2A	2B	04	86	AB	20	09	CE	00	28	C6
0110	03	8D	77	86	AD	8D	75	86	B0	8D	71	86	AE	8D	6D	7A
0120	00	2B	2A	0C	86	04	9B	2B	2A	0A	8D	62	96	2B	20	F2
0130	8D	5E	20	F8	CE	00	17	DF	23	CE	00	28	C6	03	8D	52
0140	7F	00	1A	CE	00	17	C6	03	8D	7A	8D	4A	7C	00	2B	27
0150	09	CE	00	1A	C6	04	8D	3C	20	F2	86	07	97	20	96	1A
0160	27	11	96	1A	8B	B0	8D	24	7A	00	20	27	59	8D	27	20
0170	F1	20	8D	7A	00	1B	7D	00	19	26	ED	7D	00	18	26	E8
0180	7D	00	17	26	E3	7F	00	1B	20	DE	20	5C	20	6B	20	5A
0190	20	5A	20	67	20	67	7F	00	1A	CE	00	13	DF	23	CE	00
01A0	17	C6	04	8D	ED	CE	00	17	C6	04	8D	18	CE	00	17	C6
01B0	04	8D	11	CE	00	13	DF	23	CE	00	17	C6	04	8D	67	CE
01C0	00	17	C6	04	20	64	86	C5	8D	C2	7D	00	1B	2B	04	86
01D0	AB	20	05	70	00	1B	86	AD	8D	B2	5F	96	1B	80	0A	2B
01E0	0F	97	1B	5C	20	F7	20	89	20	5C	20	60	20	5C	20	A6
01F0	17	8B	B0	8D	04	96	1B	8B	B0	20	45	20	45	20	29	CE
0200	00	10	C6	0C	6F	00	08	5A	26	FA	8D	38	81	AB	27	06
0210	81	AD	26	06	97	10	8D	28	8D	2A	81	8F	26	10	86	BC
0220	8D	1E	8D	70	20	D9	20	6E	20	6E	20	6E	20	D1	81	AE
0230	26	1E	7D	00	12	26	5B	7F	00	20	97	12	8D	02	20	D8
0240	20	5A	20	5A	20	5A	20	5C	20	9C	20	5E	20	5A	20	9E
0250	81	C5	26	5A	8D	EA	8D	48	81	AB	27	06	81	AD	26	06
0260	97	11	8D	DC	8D	3A	81	8F	27	B4	81	B0	2B	42	81	BA
0270	2A	42	84	0F	16	CE	00	1B	86	03	A1	00	2B	14	A6	00
0280	0C	69	00	69	00	AB	00	49	1B	A7	00	86	B0	1B	20	D2
0290	20	86	20	49	20	7A	20	7C	20	7C	20	7C	20	7C	20	7C
02A0	20	7C	20	88	20	7E	20	A0	20	7E	20	66	20	A0	81	B0
02B0	2B	2B	81	BA	2A	27	16	86	F8	95	19	26	D3	17	8D	5A
02C0	7C	00	20	C4	0F	37	8D	86	CE	00	13	6F	02	6F	01	32
02D0	A7	00	DF	23	CE	00	17	C6	03	8D	39	20	B3	7D	00	10
02E0	27	07	CE	00	17	C6	03	8D	3B	7F	00	16	CE	00	27	DF
02F0	23	CE	00	16	C6	04	8D	24	C6	17	D7	2B	8D	66	96	11
0300	27	03	70	00	1B	96	12	27	21	4F	90	20	20	1C	20	28
0310	20	54	20	3C	20	67	20	67	20	67	20	4E	20	65	20	50
0320	20	80	20	82	20	5F	20	84	20	0E	9B	1B	97	1B	2B	1B
0330	26	01	39	8D	03	26	FC	39	86	04	97	33	86	50	97	32
0340	4F	97	31	97	30	8D	7A	7A	00	1B	39	8D	03	26	FC	39
0350	86	FD	97	33	86	66	97	32	97	31	86	67	97	30	8D	61
0360	7C	00	1B	39	20	5D	86	A0	8D	00	36	BD	E1	D1	32	39
0370	86	3C	B7	80	07	BD	E1	AC	8A	80	39	20	A5	20	72	20
0380	74	20	74	20	6E	20	72	20	97	86	8D	8D	DD	86	8A	8D
0390	D9	8D	8D	8D	D1	CE	00	1C	DF	23	CE	00	28	C6	04	8D
03A0	E2	8D	CD	81	AB	26	06	8D	2E	8D	52	20	20	81	AD	26
03B0	06	8D	24	8D	4A	20	16	81	D8	26	0A	8D	1A	8D	3C	20
03C0	0C	20	38	20	3E	81	AF	26	08	8D	0C	8D	34	8D	AC	20
03D0	B8	81	8F	26	CC	20	B2	8D	91	8D	8B	8D	AA	8D	87	86

```
03E0  BD 8D 87 8D 81 CE 00 30 DF 23 CE 00 1C C6 04 20
03F0  02 20 56 20 72 20 66 20 58 20 4A 20 4A 20 4C 20
0400  4C 20 4C CE 00 25 96 2A 2B 04 6F 00 20 09 A7 00
0410  C6 04 CE 00 27 8D 77 CE 00 2A C6 04 6D 00 26 08
0420  09 5A 26 F8 7F 00 2B 39 CE 00 27 C6 04 8D 22 6D
0430  00 2B 04 6A 01 20 F1 CE 00 2A C6 03 8D 1F 7D 00
0440  25 27 E4 C6 03 20 47 20 68 20 2E 20 5E 20 5E 20
0450  5E 0C 69 00 5A 26 01 39 08 20 F7 20 A6 0C 66 00
0460  5A 26 01 39 09 20 F7 DF 21 A6 00 DE 23 A7 00 08
0470  DF 23 DE 21 08 5A 26 EF 39 0C A6 00 DF 21 DE 23
0480  A9 00 08 DF 23 DE 21 A7 00 08 5A 26 ED 39 60 00
0490  5A 26 01 39 08 24 F7 63 00 20 F5 20 B4 20 B3 20
04A0  BA 20 BA 20 B9 20 C0 20 D0 20 E3 20 71 20 71 20
04B0  6B 8D 7F 96 33 9B 2B 4C 97 2B 86 17 97 20 CE 00
04C0  2A C6 03 8D 98 24 0C CE 00 2D DF 23 CE 00 35 C6
04D0  06 8D A6 CE 00 3A C6 06 8D 83 7A 00 20 26 DF CE
04E0  00 3A C6 06 8D BB CE 00 37 A6 00 49 2A 12 C6 03
04F0  86 40 AB 00 A7 00 08 86 00 A9 00 5A 26 F6 A7 00
0500  CE 00 27 DF 23 CE 00 37 C6 04 20 16 20 8D 20 8D
0510  20 8D 20 8D 20 8D 20 8D 20 8D 20 8D 20 62 20 5C
0520  20 5C 8D 81 8D EA 96 26 26 07 CE 00 28 C6 03 8D
0530  E9 39 C6 08 CE 00 34 6F 00 08 5A 26 FA C6 04 CE
0540  00 2C 6F 00 08 5A 26 FA 86 01 97 26 96 2A 2A 0A
0550  7A 00 26 CE 00 28 C6 03 8D C0 7D 00 32 2B 01 39
0560  7A 00 26 CE 00 30 C6 03 20 B0 20 A0 20 A0 20 B2
0570  20 9E 20 9E 20 9E 20 9E 20 9E 20 9E 20 7A 20 7A
0580  8D B0 7D 00 2A 27 1E 96 33 90 2B 4C 97 2B 86 17
0590  97 20 8D 68 2B 14 CE 00 30 DF 23 CE 00 34 C6 03
05A0  8D D4 0D 20 06 86 BF 7E E1 D1 0C CE 00 38 C6 03
05B0  8D BA CE 00 30 C6 03 8D B1 7A 00 20 26 D4 8D 3C
05C0  2B 1E 86 01 9B 38 97 38 86 00 99 39 97 39 86 00
05D0  99 3A 97 3A 2A 0A CE 00 3A C6 03 8D 95 7C 00 2B
05E0  CE 00 27 DF 23 CE 00 37 C6 04 20 82 20 82 20 82
05F0  20 82 20 82 20 82 20 82 20 37 20 2E CE 00 34 DF
0600  23 CE 00 28 C6 03 8D EA CE 00 34 DF 23 CE 00 30
0610  C6 03 0C A6 00 DF 21 DE 23 A2 00 A7 00 08 DF 23
0620  DE 21 08 5A 26 ED 7D 00 36 39 CE 00 28 C6 03 8D
0630  C5 7D 00 2A 26 0C CE 00 28 DF 23 CE 00 30 C6 04
0640  20 B0 7D 00 32 26 09 39 20 A2 20 A2 20 A2 20 A4
0650  CE 00 2B A6 00 91 33 27 2C 40 9B 33 2A 01 40 81
0660  18 2B 08 E6 00 96 33 10 2A CC 39 96 33 E6 00 10
0670  16 2B 0A CE 00 2B 8D 2B 5A 26 F8 20 08 CE 00 33
0680  8D 21 5C 26 F8 7F 00 27 7F 00 2F CE 00 2B 8D 13
0690  CE 00 33 8D 0E CE 00 2F DF 23 CE 00 27 C6 04 8D
06A0  AD 20 A5 6C 00 09 17 C6 04 6D 00 2B 04 8D 9B 20
06B0  03 0D 8D 98 16 39
```

ACCMIN	040E	FOPMSW	0032
ACCSET	0437	FOPNSW	0031
ACNONZ	0428	FPACCE	002B
ACZERT	0417	FPADD	0631
ADDER	0479	FPCONT	0389
ADDEXP	04B3	FPD10	0350
ADOPPP	04C7	FPDIV	0580
AHEAD1	0115	FPINP	01FF
AHEAD2	01D8	FPLSW	0028
BRING1	06B1	FPLSWE	0027
CKEQEX	0650	FPMSW	002A
CKSIGN	0532	FPMULT	04B1
CLMOR1	0537	FPNORM	0403
CLMOR2	0542	FPNSW	0029
CLRNXT	0204	FPOUT	0100
CNTR	0020	FPSUB	062A
COMPEN	014C	FPX10	0338
COMPLM	048E	FSHIFT	06A9
CONTNU	0494	INEXPS	0011
CROUND	04F4	INMTAS	0010
DECBIN	0196	INPRD1	0012
DECEXD	0130	INPUT	0370
DECEXT	0122	IOEXP	0016
DECOUT	0134	IOEXPD	001B
DECRDG	0168	IOLSW	0013
DECREP	012C	IOMSW	0015
DERROR	05A5	IONSW	0014
DIVIDE	0592	IOSTR	0017
DOMOR	0490	IOSTR1	0018
DVEXIT	05E0	IOSTR2	0019
ECHO	036A	IOSTR3	001A
ENDINP	02DD	LINEUP	066B
ERASE	021E	LOOK0	041C
EXECHO	0262	MCAND0	002C
EXMLDV	0522	MCAND1	002D
EXOUTN	01D3	MCAND2	002E
EXPFIX	0333	MINEXP	034B
EXPINP	0264	MORACC	0673
EXPOK	032A	MORRTL	0458
EXPOUT	01C6	MORRTR	0464
FINAL	03CD	MOVEIT	0467
FINPUT	02E9	MOVOP	0636
FNDEXP	0254	MULTEX	0531
FOLSWE	002F	MULTIP	04BE
FOPEXP	0033	NADOPP	04D3
FOPLSW	0030	NEGFPA	0550

NEGOP	0560	SETSUB	05FC
NINPUT	0218	SFNDXP	02AE
NOEXPS	0266	SHACOP	0685
NOGO	05AA	SHIFTO	067D
NONZAC	0642	SHLOOP	06A3
NORMEX	0427	SIGNS	0026
NOTADD	03AD	SKPNEG	065F
NOTDIV	03D1	SPACES	0366
NOTMUL	03C5	SPRIOD	0250
NOTPLM	021A	SUB12	01DD
NOTSUB	03B7	SUBBER	0612
NVALID	03A1	SUBEXP	0587
OPERAT	03D7	TEMP1	0021
OPSGNT	055A	TEMP2	0023
OUTDGS	0162	TOMUCH	01F0
OUTDIG	015A	TPEXP	001F
OUTNEG	010C	TPLSW	001C
PERIOD	0232	TPMSW	001E
POSEXP	0305	TPNSW	001D
PREXFR	0500	TSIGN	0025
QUOROT	05AB	WORK0	0034
RESCNT	06B4	WORK1	0035
ROTATL	0451	WORK2	0036
ROTATR	045D	WORK3	0037
ROTL	0452	WORK4	0038
ROTR	045E	WORK5	0039
SECHO	0216	WORK6	003A
SERASE	022E	WORK7	003B
SETDCT	058E	ZERODG	0173
SETMCT	04BA		

6281-8
$12.95

6800
SOFTWARE GOURMET GUIDE & COOKBOOK

ROBERT FINDLEY

An essential guide to machine language programming for 6800-based microcomputers. The book is designed to take BASIC language programmers into the realm of machine language programming stressing the advantages and additional programming power available to the user with 72 fundamental instructions.

There are several essential elements in the structure of the Motorola 6800 CPU with which the programmer must become thoroughly familiar. These include the program counter, the two accumulators, the index register, stack pointer, memory, and the status flags. The author stresses addressing modes that provide a versatility for creative programming. General purpose subroutines, conversion routines, and input/output processing are carefully and clearly explained and illustrated with examples.

Other Books of Interest...

THE 8086/8088 PRIMER: An Introduction to Their Architecture, System Design, and Programming Second Edition

STEPHEN P. MORSE

This revised edition has been updated to provide novices and professionals alike with a thorough introduction to Intel's 16-bit 8086 and 8088 microprocessors.

After a general introduction to computers and microprocessors, with emphasis on the 8086, the book discusses architecture: the machine organization of the 8086/8088, covering register and memory structure, addressing modes, and the 8086/8088 instruction set.

The section on system design features a new chapter highlighting the 8088 used in the IBM personal computer. The book concludes with a discussion of programming. In addition to chapters on a low-level programming language, ASM-86, and a high-level language, PL/M-86, a new chapter examines the Pascal language. #6255-9, paper, 288 pages.

PROGRAMMER'S GUIDE TO THE 1802 (With an Assembler for Your Machine)

TOM SWAN

Here's an assembly language primer that has an assembler! Coverage includes everything from the binary number system and the fundamentals of machine language to the development of a working 1802 assembler. Simply written in nontechnical language, the text is intended for the beginner but contains information that will be appreciated by experts. #5183-2, paper, 168 pages.

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

ISBN 0-8104-6281-8